E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Diseño y desarrollo de una Smart City App

Grado en Ingeniería Informática

Trabajo Fin de Grado

Asier Guilló Garitano, autor

José Javier Astrain Escola, director

Pamplona, 8 de junio de 2017

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Agradecimientos

En primer lugar, quisiera agradecer todo su esfuerzo y dedicación a mi tutor de este Trabajo de Fin de Grado, José Javier Astrain Escola. Gracias por todas las horas que has invertido en ayudarme a terminar este trabajo.

Por supuesto, a mis padres y mi hermana, por haber sido siempre mi mayor apoyo, hasta cuando las cosas no iban como deberían.

A la Universidad Pública de Navarra y en especial a todos los profesores que han tomado parte en mi formación y que en mayor o menor medida han contribuido en ella.

Al Ayuntamiento de Pamplona, que a través del Instituto Smart Cities me ha permitido participar en un proyecto de gran envergadura como este.

Por último, a mis compañeros y amigos de promoción. Por hacer estos años mucho más llevaderos, haciendo que las largas horas de biblioteca pasaran volando y ayudarme siempre que les ha sido posible. En especial a Dani por aguantar mis tostones estos últimos meses y ayudarme a sacar el trabajo adelante.

Resumen

El propósito de este Trabajo de Fin de Grado consiste en desarrollar una aplicación móvil para el sistema operativo Android y una red de sensores que recoja información sobre la contaminación de la ciudad con el objetivo de ofrecer a los ciudadanos de Pamplona una aplicación de ciudad inteligente.

Para ello se ha desarrollado una aplicación haciendo uso del lenguaje de programación Kotlin que ofrece algunas ventajas respecto a los lenguajes tradicionales usados para desarrollar en esta plataforma. Entre la información más relevante que muestra la aplicación se encuentra la información del servicio de alquiler de bicicletas, del servicio de recarga de vehículos eléctricos y de los valores obtenidos por la red de sensores desarrollada.

La red de sensores está compuesta por nodos que periódicamente envían los datos recogidos por los sensores a un servidor que los almacena y se los ofrece a modo de API REST a la aplicación.

Palabras Clave

Kotlin, Smart City, Pamplona, Android, Waspmote, Raspberry Pi, Stardust

Abstract

The purpose of this Final Degree Project is to develop a mobile application for the Android operating system and a sensor network that collects information on the city's pollution in order to offer the citizens of Pamplona a smart city application.

For this reason, an application has been developed using the Kotlin programming language that offers some advantages over the traditional languages used to develop in this platform. Among the most relevant information that the application provides, we found information about the bicycle rental service, the electric vehicle recharging service and the values obtained by the developed sensor network.

The sensor network consists of nodes that periodically send the data collected by the sensors to a server that stores them and offers them as a REST APIs to the application.

Keywords

Kotlin, Smart City, Pamplona, Android, Waspmote, Raspberry Pi, Stardust

Índice

Índice de Ilustraciones

1. Introducción

El proyecto surge de la selección de Pamplona por parte de la Comisión Europea como una de las tres ciudades europeas para el desarrollo de un proyecto piloto sobre ciudades inteligentes. Pamplona fue seleccionada junto con Tampere (Finlandia) y Trento (Italia) entre más de 17 propuestas, en el marco del proyecto Stardust [1]. La propuesta de la ciudad navarra fue un proyecto coordinado por el Centro Nacional de Energías Renovables (CENER) cuya duración ascenderá a los 5 años y en cuyo desarrollo participarán un total de 29 socios entre los que se encuentran empresas públicas y privadas, universidades y centros tecnológicos. El proyecto, denominado Stardust, cuenta con una serie de objetivos que se detallarán más adelante relacionados con la conectividad y el Internet de las cosas (IoT).

1.1 Introducción

El objetivo del proyecto Stardust es aprovechar las tecnologías de la información y los datos que se podrían recoger de la ciudad para generar información que permita tomar mejores decisiones y facilitar la gestión de la ciudad, mejorando así la calidad de vida de los ciudadanos que la habitan. Tal y como se indica en la estrategia del proyecto [2], para llevar a cabo esta tarea se crearán 5 grupos de trabajo junto con los objetivos de cada uno:

Innovación Social

- Accesibilidad de la ciudad mediante ascensores urbanos y rampas.
- Transparencia mediante un portal de acceso a datos públicos del Ayuntamiento.
- Accesibilidad a la información sobre la ciudad mediante su página web y redes sociales.
- Disponer de atención ciudadana telefónicamente.
- Promover la participación ciudadana.
- Ofrecer conexión Wifi gratuita en espacios públicos.

- Ofrecer un sistema de alertas SMS en caso de emergencias.
- Controlar el aforo de la ciudad mediante espiras electromagnéticas.
- Controlar el número de vehículos en la ciudad mediante detectores de matrículas.
- Vigilar mediante cámaras los espacios públicos y controlar el tráfico.
- Detectar vehículos robados mediante cámaras en los vehículos de la policía municipal.

Energía

- Instalar estabilizadores de tensión y reductores de flujo con el fin de obtener un ahorro energético del 40% (iluminación inteligente).
- Eficiencia en los gastos energéticos de los edificios públicos.
- Instalar contadores inteligentes de electricidad.
- Instalar lamparas LED en los semáforos con el fin de obtener un ahorro energético del 88%.
- Promover el uso de energías renovables en edificios municipales.

Medio ambiente

- Controlar el cambio climático y la calidad ambiental.
- Instalar equipos de riego automático.
- Realizar recogidas de residuos selectivas en zonas de la ciudad.
- Promover el reciclaje y contribuir al desarrollo sostenible.

Movilidad urbana

- Gestionar adecuadamente el tráfico de la ciudad mediante semáforos, cámaras, radares e indicadores de velocidad.
- Controlar el acceso a ciertos puntos de la ciudad mediante pivotes neumáticos.
- Gestionar los aparcamientos de la ciudad con pantallas informativas de ocupación distribuidos por la ciudad.
- Promover el servicio de compartir coche.
- Promover el uso de vehículos eléctricos mediante el estacionamiento gratuito en zonas de pago.
- Promover el servicio de alquiler de vehículos eléctricos.

- Promover el servicio de alquiler de bicicletas.
- Informar de los tiempos de llegada del transporte urbano en las marquesinas.

Gobierno

- Ofrecer servicios de la administración mediante la administración electrónica.
- Ofrecer mediante quioscos en la ciudad la información y servicios del portal web.

Además de todos los objetivos citados anteriormente, se desarrollará una plataforma tecnológica que recogerá, analizará y filtrará la gran cantidad de datos que generará la ciudad. Estos datos serán accesibles por un lado para las administraciones públicas, lo que permitirá mejorar los servicios ciudadanos, y por el otro para los ciudadanos.

Este trabajo de fin de grado comienza con la idea de formarme en un nuevo lenguaje de programación para el desarrollo de aplicaciones en Android (Kotlin) y la posibilidad de trabajar en un proyecto innovador. Con el fin de promover la participación ciudadana y el acceso a la información de la ciudad, se plantea una aplicación móvil que reúna toda la información relativa a la movilidad de la ciudad, así como otras informaciones relacionadas con la movilidad como son las atmosféricas (temperatura, humedad, radiación solar…) o contaminación.

1.2 Antecedentes

El proyecto Stardust, como se ha comentado con anterioridad, tiene como fin desarrollar una ciudad inteligente que permita mejorar la calidad de vida de los ciudadanos. El Instituto para la Diversificación y el Ahorro de la Energía (IDAE) [2] define como una ciudad inteligente aquella que respete el medio ambiente, que utilice las Tecnologías de la Información y las Comunicaciones (TIC) para su gestión y cuyo desarrollo sea sostenible. Según un estudio de la Organización de Consumidores y Usuarios (OCU) del año 2012 [2], Pamplona era la ciudad de tamaño medio con la mejor calidad de vida de España y en cuanto a sostenibilidad, un estudio realizado en el año 2012 por Siemens [2] situó a Pamplona como la quinta ciudad más

sostenible de España, teniendo en cuenta que solo era superada, a excepción de Vitoria, por grandes ciudades como Madrid, Bilbao o Barcelona.

Ilustración 1. Clasificación de ciudades más sostenibles (2012)

Este estudio tuvo en cuenta factores como las emisiones CO2, la calidad del aire, la utilización energética, el uso del agua, la generación y gestión de residuos, la movilidad y la construcción para valorar el nivel de sostenibilidad de la ciudad. Además, Pamplona fue destacada como aspirante a Smart City en el año 2012 por un estudio de la consultora IDC [2].

Ilustración 2. Ranking de Ciudades Inteligentes de IDC (2012)

Cabe destacar que Pamplona fue destacada como aspirante a pesar de no haber realizado ningún plan específico en este sentido. Durante los últimos años la capital navarra ha tenido un proceso de mejora continuado en lo que a ciudad inteligente se refiere, pero siempre mediante una mejora sectorial y no global como se plantea en este proyecto. El ayuntamiento de la ciudad ha tenido un papel clave en este proceso y parte del éxito viene dado por la transparencia de esta institución. En el informe realizado por el Índice de Transparencia de los Ayuntamientos (ITA) en el año 2012 [2], Pamplona obtuvo un sobresaliente 93,8 de puntuación con un alto nivel de evolución como se muestra en el siguiente gráfico.

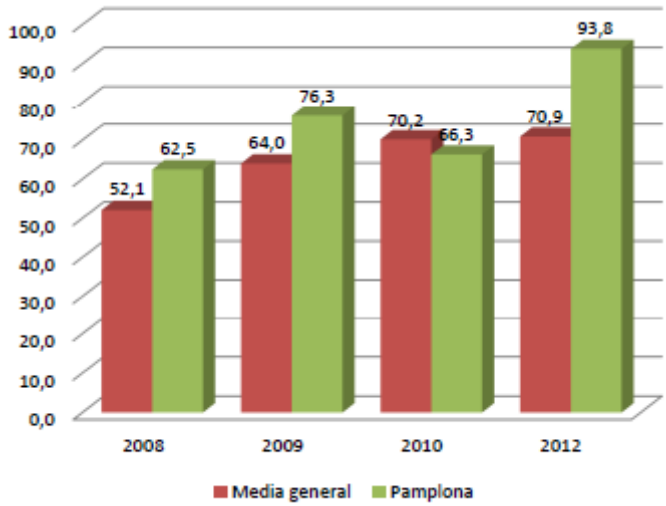

Ilustración 3. Informe transparencia de ITA (2012)

El proyecto STARDUST [1] es un proyecto sobre Smart Cities que forma parte del Programa Marco de la Unión Europea. Este programa, denominado Horizonte 2020 [3], tiene como objetivo apoyar a proyectos de investigación, desarrollo e innovación (I+D+I). Este es el octavo Programa Marco presentado por la UE y abarca el periodo de los años 2014-2020. Uno de sus principales objetivos es investigar sobre los problemas que afectan a los ciudadanos europeos. Es por esto por lo que el proyecto STARDUST cobra bastante relevancia dentro del citado programa de la UE. Este proyecto [1] tiene como fin reunir a ciudades europeas que sean eficientes, inteligentes y orientadas a los ciudadanos para difundir la innovación en las ciudades de la UE. Para ello se seleccionaron tres “ciudades faro” (Pamplona, Trento y Tampere) que servirán de modelo para que las “ciudades seguidoras” (Cluj-napoca, Derry, Kozani, Litomerice) repliquen la

estrategia y las acciones llevadas a cabo en las primeras. Estas ciudades formarán una constelación de ciudades inteligentes e innovadoras que dan forma al logo del proyecto.

Ilustración 4. Proyecto STARDUST

Los objetivos de este proyecto son los siguientes:

- Crear “islas de innovación” que demuestren soluciones escalables y rentables.
- Crear ecosistemas inteligentes que hagan uso del nuevo paradigma económico en las ciudades europeas que se basa en la ecoinnovación, la competitividad del mercado, el bajo consumo de carbono y la promoción de una economía circular.
- Crear y desplegar plataformas TIC que permita abordar los problemas de manera efectiva.

Organizar y fomentar las soluciones de las “ciudades faro” para que sean replicadas en las “ciudades seguidoras”.

1.3 Objetivos

Tras poner en contexto el proyecto STARDUST y los objetivos que éste tiene, y tras reunirse con los responsables municipales del proyecto Stardust, se planteó crear una aplicación relacionada con la movilidad de la ciudad. Esta aplicación englobaría todos los servicios de movilidad con los que contaba la ciudad, entre los cuales podríamos nombrar las villavesas, las bicicletas de alquiler, los taxis, los vehículos eléctricos, etc. El objetivo principal era hacer uso de estos servicios para minimizar el uso de los

coches particulares, de manera que se consiguiera reducir los niveles de contaminación y mejorar la sostenibilidad de la ciudad.

Los objetivos que fijé para el desarrollo de esta aplicación fueron los siguientes:

- Desarrollada en Kotlin.

  La aplicación debía de ser desarrollada en Kotlin. Este lenguaje, del que se hablara más en profundidad más adelante, surgió en el año 2012 pero no fue hasta el año 2017 cuando alcanzó la popularidad. Su principal objetivo es mejorar el lenguaje de programación Java. Requiere una fase de aprendizaje por lo que el objetivo es formarse en esta tecnología y ser capaz de desarrollar la aplicación en su totalidad mediante esta.

- Compatible con la mayoría de los dispositivos Android.

  Se usará la versión 15 de SDK (Software Development Kit). Esto indica la versión de las herramientas que se usarán para el desarrollo de la aplicación e indica a partir de qué versión de Android se podrá usar la aplicación. En este caso el SDK 15 indica que la aplicación se podrá instalar en todos los dispositivos que contengan de Android 4.0.3 (IceCream Sandwich) en adelante. Teniendo en cuenta las cuotas de mercado actuales [4], eso significa que será compatible con un 99% de los dispositivos Android, porcentaje que habrá subido debido al alza de la versión 8.0 (Oreo) en los últimos meses.

- Interfaz de usuario sencilla e intuitiva.

  El diseño de las pantallas de la aplicación debe ser intuitiva y fácil de usar. Esta aplicación está pensada para gente de cualquier edad, por lo que las pulsaciones que una persona haga para llegar a donde quiere deben ser las mínimas posibles y debe estar bien identificado a dónde se está yendo o qué información se va a recibir.

- Información acerca del servicio de alquiler de bicicletas.

La aplicación debe mostrar la información acerca del servicio de alquiler de bicicletas como son las paradas, las bicicletas disponibles, la posición de las estaciones y demás información relevante sobre dicho servicio.

- Información acerca del servicio de recarga de vehículos eléctricos.

Se deberá poder visualizar toda la información relativa al servicio de recarga de vehículos eléctricos. Dentro de los objetivos del proyecto STARDUST consta el despliegue de más puntos de recarga con el fin de promover la compra y el uso de este tipo de vehículos que además cuentan con ventajas en temas de estacionamiento. La información a ofrecer sería la disponibilidad de la estación, así como su posición, velocidad de recarga, etc.

- Información sobre cómo llegar de un sitio a otro eficientemente.

Se debe mostrar la mejor opción para viajar de un punto de la ciudad a otro usando el servicio más rápido y sostenible. Esta información deberá llegar a la aplicación de la plataforma Smart que se planea desarrollar y ser mostrada por la aplicación de la manera más adecuada, ofreciéndole al usuario todas las opciones posibles.

- Información de la ciudad.

Al ser una aplicación sobre la ciudad, será necesario mostrar algún tipo de información relativa a la ciudad. Se plantea mostrar parámetros ambientales como temperatura o contaminación, y cualquier información que pueda ser interesante para el usuario.

- Interacción con la ciudadanía.

La colaboración ciudadana debe ser un factor importante para el éxito de un desarrollo adecuado de la ciudad. Por ello se debe implementar algún tipo de sistema para facilitar la comunicación entre los usuarios y su ayuntamiento. Este sistema podrá facilitar la gestión de indecencias o sugerencias, de manera que se podrían

clasificar por tipos y categorizarlas para hacérselas llegar a la persona adecuada para ser resueltas.

El hecho de ser un proyecto con una duración de 5 años hace que la plataforma a la cual la aplicación se tendría que conectar para recoger todos los datos no vaya a estar completamente desarrollada y operativa para la finalización de este trabajo de fin de grado. En resumidas cuentas, esto significa que la aplicación que se desarrollará para el trabajo no será plenamente funcional. Teniendo esto en cuenta, se pretende que esta aplicación sea un demostrador tecnológico, es decir, que dé una idea de lo que debería ser una aplicación de una ciudad inteligente, algunos de los servicios que deberá ofrecer y presentar un diseño que bien podría valer para la aplicación final.

Aparte del tema de la movilidad urbana, se quiere dar al trabajo otro enfoque más centrado en el tema medioambiental. Por ello se empleará una red de sensores que recogerán datos relevantes del aire, por ejemplo, la temperatura, la humedad, el monóxido de carbono (CO), el dióxido de carbono (CO_2) y otros gases nocivos para el medio ambiente. El objetivo es emplear un dispositivo (mota) dotado de una placa que permita tomar datos de diferentes sensores y mediante algún sistema de comunicación envíe estos datos para su almacenamiento y posterior uso. De la misma manera se deberá poner en marcha un servidor que recoja estos datos y los almacene. Gracias a esta red de sensores se quiere dar a conocer cómo se podría monitorizar la contaminación de la ciudad poniendo sensores por toda ella. Esto ayudaría a gestionar la contaminación focalizándose por zonas y se podrían tomar decisiones más apropiadas en función de la información disponible en la correspondiente base a los datos. Por poner un ejemplo práctico, se podría cerrar alguna parte de la ciudad a vehículos con motores de combustión si la contaminación del aire superara los limites estipulados. La idea es que todos los datos se recojan, almacenen y gestionen en la plataforma que se desarrollará.

1.4 Estado del arte

Una vez explicado cuáles son los objetivos del trabajo, se mencionarán algunas aplicaciones que existen relacionadas con lo que se pretende desarrollar y cuáles son las funcionalidades que cada una ofrece. Estas

aplicaciones se podrían englobar como partes de lo que se pretende hacer, ya que cada una corresponde a un servicio de movilidad o informativo del ayuntamiento.

TuVillavesa

Esta aplicación, desarrollada por la empresa GeoActio para el servicio de villavesas de la Mancomunidad de la Comarca de Pamplona, muestra información de las líneas de villavesa existentes, así como los tiempos de llegada de cada una. Se trata de una aplicación relativamente vieja, ya que se lanzó a finales del año 2011 cuando Android se encontraba en su versión 3.0 (Honeycomb). Por lo tanto, estamos hablando de una aplicación con un aspecto desfasado, en cuanto a diseño se refiere, que no sigue las normativas del Material Design [5] o normativas de diseño de Google de hoy en día. Esta normativa define pautas o normas a la hora de realizar diseños para Android y ofrece librerías para que todos los elementos gráficos sean lo más novedosos y usables posibles. A pesar de ello sigue ofreciendo el servicio para el que fue desarrollada, que es mostrar los tiempos de llegada para cada parada, mostrar cómo llegar de un sitio a otro de Pamplona usando villavesas, mostrar información acerca de las líneas y las paradas cercanas al usuario (haciendo uso de la posición del usuario).

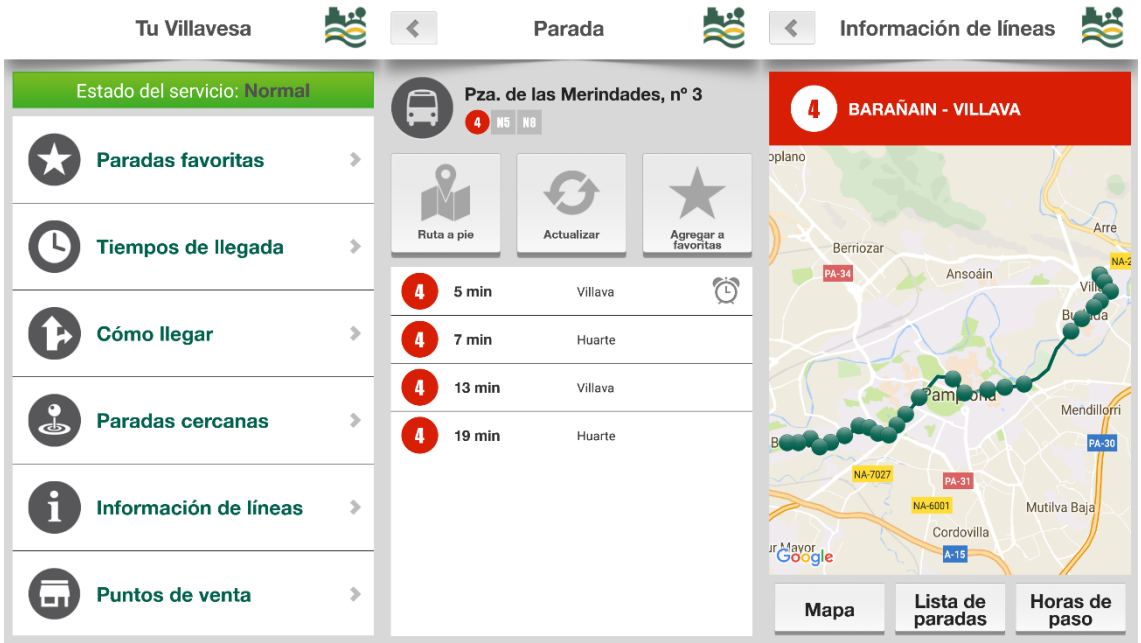

Ilustración 5. Pantallazos de TuVillavesa

Pamplona (Ayuntamiento de Pamplona)

Aplicación del ayuntamiento de la ciudad puramente informativa. Cuenta con noticias, eventos y cursos relacionados con Pamplona. Además, permite contactar con el ayuntamiento mediante un sistema de atención ciudadana basada en un formulario para sugerencias o quejas. Fue lanzada a mediados del año 2015 y tiene un diseño más moderno comparándolo con el de TuVillavesa. Tal y como se ha comentado, su finalidad es más bien informativa, aunque ofrece una parte de colaboración ciudadana que la hace interesante o al menos digna de mencionar.

Ilustración 6. Pantallazos de Pamplona

Parkplona

Esta aplicación lanzada a comienzos del 2013 tiene el fin de encontrar aparcamiento en la ciudad. Permite localizar todos los aparcamientos disponibles, tanto gratuitos como de pago. Ofrece información en tiempo real del número de plazas libres (si el parking lo monitoriza), la capacidad máxima, su posición y cómo llegar, el horario de apertura, las tarifas, etc. También muestra la posición en un mapa de las estaciones de alquiler de bicicletas y los puntos de recarga de vehículos eléctricos. Para finalizar, se puede consultar el Depósito Municipal de Vehículos introduciendo la matricula, suscribirse al servicio de alertas SMS o ver las cámaras de tráfico de la ciudad. Al igual que la aplicación de las villavesas se nota que tiene un diseño más antiguo, aunque también es la que más se asemeja a lo que se quiere hacer en este trabajo.

Ilustración 7. Pantallazos de Parkplona

Nbici

Se trata de una aplicación web que muestra el estado de las estaciones de bicicletas de Pamplona en tiempo real. Se puede consultar la cantidad de bicis disponibles en cada una y, aunque muestra la situación de una manera gráfica, la posición de las bicicletas no se corresponde con la posición real de estas. La web además muestra toda la información relacionada con este servicio, desde cómo registrarse hasta los horarios y condiciones de uso.

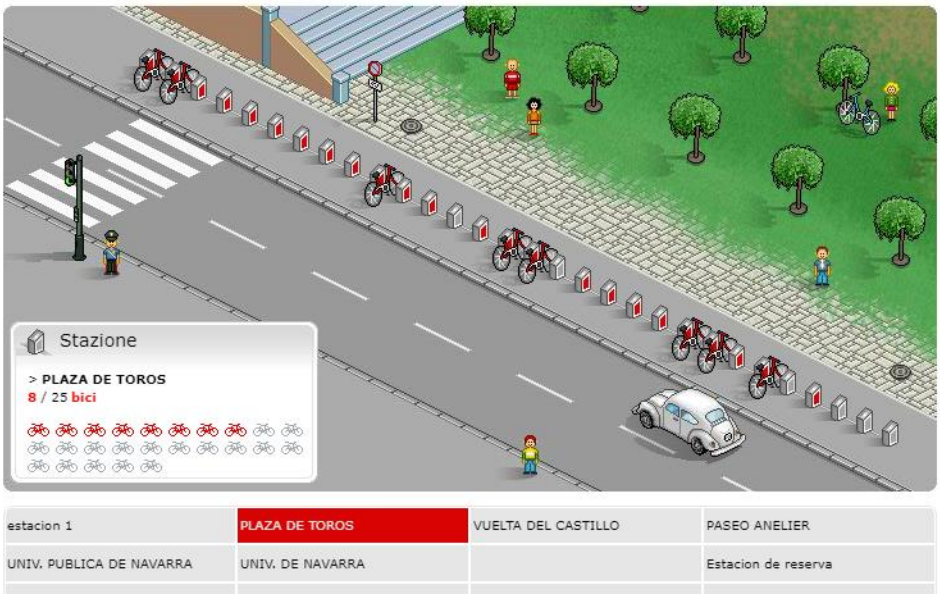

Ilustración 8. Pantallazo de Nbici

Aplicación de bicicletas robadas

Esta aplicación web fue anunciada [6] a mediados de mayo. Su objetivo es facilitar la recuperación de sus bicicletas a las personas a las que les han robado la bicicleta. Los denunciantes obtendrán acceso a la aplicación mediante el número de denuncia y una contraseña que se les facilitará mediante SMS o correo electrónico. Este acceso tendrá una caducidad de 3 meses y permitirá a los usuarios visualizar las bicicletas que coincidan con la marca y con la fecha en la que fue sustraída.

Aplicación de incidencias que impidan la accesibilidad

Esta aplicación, al igual que la anterior la anterior, fue anunciada [7] a mediados de mayo. Mediante esta aplicación el Ayuntamiento de Pamplona conocerá, mediante la colaboración ciudadana, las incidencias o desperfectos que afecten a la accesibilidad de la ciudad, dificultando o impidiendo la movilidad de las personas.

1.5 Propuesta

Se propone desarrollar una aplicación Smart City para dispositivos Android, así como una red de sensores que analicen la calidad del aire. Será necesario desarrollar también un sistema para la recolección de los datos que los sensores captarán.

Para ello se estudiará la información que necesita un ciudadano de la ciudad para posteriormente desarrollar una aplicación que reúna todo lo necesario. Se analizará cada servicio individualmente y se planteará qué datos serían necesarios tener de cada uno de ellos. Además, todo ello deberá quedar reflejado en el prototipo desarrollado.

A partir de ahora se le denominará prototipo a la aplicación que se desarrollará, ya que a pesar de ser desarrollada para datos que podrían ser reales no contará con datos reales en ningún momento. Todos los datos serán introducidos manualmente a modo de mockup y no recibidos de un servidor, serán datos sintéticos. En un futuro, la aplicación deberá interactuar con la plataforma que se está desarrollando y que contendrá todos los datos para su óptimo funcionamiento. De esta manera se dispondrá de información en tiempo real de la situación de los puntos de

recarga, estaciones de alquiler de bicicletas o contaminación de la ciudad que podrá ser mostrada en la aplicación. La aplicación deberá contar con alguna pantalla o sistema de interacción con los ciudadanos, aunque las quejas o incidencias no vayan a la plataforma que en un futuro estará disponible para ser analizadas correctamente.

En cuanto a la contaminación aérea se propone integrar un sensor que permita recoger datos del aire y sean mandados a un servidor para su almacenamiento y distribución. Este servidor llegado el momento deberá ser sustituido por la ya mencionada plataforma, hecho que no ocurrirá antes de la finalización del trabajo de fin de grado. Es por ello por lo que habrá que poner en marcha un servidor con todo lo necesario para emular lo que en un futuro deberá suceder.

Esta red de sensores deberá ser probada obteniendo datos reales para su posterior análisis. Se analizarán los datos obtenidos para sacar conclusiones acerca de la contaminación, comparando los datos de más de una ubicación valor por valor. Se deberá analizar cómo afecta el entorno en mayor o menor medida a los resultados obtenidos.

2. Análisis

En este capítulo se analizarán los objetivos de este proyecto y los requisitos que deberá cumplir. Los objetivos fueron analizados en un primer momento por mí y una vez se tenía un primer análisis de los objetivos que se querían cumplir con la aplicación se programó una reunión con el Ayuntamiento de Pamplona. En esta reunión se expuso el análisis realizado y se terminaron de fijar los objetivos conjuntamente. Primero se explicará el análisis de los objetivos que se realizó en la reunión y posteriormente los requisitos funcionales y no funcionales que fijé. Para finalizar se analizará la estructura general del proyecto.

2.1 Objetivos

Para empezar, se van a definir los objetivos específicos de la aplicación y qué se pretende conseguir con cada uno de ellos. Es necesario analizar cada uno de los objetivos para saber qué información se manejará y cómo se debe mostrar esta información al usuario.

- Diseño del interfaz de usuario

  Se busca un diseño de aspecto atractivo y moderno capaz de ser entendible por usuarios de todas las edades. La aplicación debe seguir el principio de los mínimos clics posibles. Este principio básico de usabilidad define que un usuario debe poder acceder a toda la información de la aplicación realizando una cantidad de clics pequeña, que normalmente suele ser como máximo 3 (Regla de los tres clics). Como ya hemos visto en el capítulo anterior una aplicación realizada hace más de 5 años queda obsoleta en lo que a diseño se refiere y deja de ser atractiva para los usuarios. Hoy en día, los diseños deben ser limpios, sencillos y los más minimalistas posibles. Debido a la evolución de los dispositivos móviles y su tremendo auge en los últimos años, el diseño de interfaces se ha convertido una parte importante del desarrollo para diferenciarse de la competencia dentro de un mercado tan extenso como el de las aplicaciones

móviles. Para esta aplicación en concreto se deberá seguir las pautas que marca el Material Design que a continuación se explicará.

Material Design [5] es una normativa desarrollada por Google que fue lanzada en el año 2014 e integrada en el sistema Android 5.0 Lollipop lanzado ese mismo año, así como en todas las aplicaciones de la propia empresa. Mediante APIs permiten a los desarrolladores incorporar esta filosofía a sus aplicaciones, sobre todo en desarrollos para Android, aunque también es aplicable a aplicaciones de escritorio o aplicaciones web. Su objetivo es ofrecer un diseño innovador que permita ofrecer la misma experiencia en cualquier plataforma o tamaño de dispositivo, lo que hoy en día se conoce como un diseño web adaptable (Responsive Web Design). Esta normativa se rige por tres principios:

- Material is the metaphor

  Las formas y bordes ayudan al usuario a comprender la funcionalidad del elemento. La luz, superficie y el movimiento son claves para transmitir cómo el objeto se mueve e interactúa con los demás.
- Bold, graphic, intentional

  Las tipografías, espacios, escalas, colores e imágenes crean significado además de agradar al usuario. Los elementos deben sumergir al usuario en la experiencia de uso gracias a una acertada interfaz gráfica.
- Motion provides meaning

  Como dice el propio principio, el movimiento ofrece significado. Cada movimiento o acción debe ser eficiente y coherente, ofreciendo la información adecuada al usuario para mantener su atención.

- Servicio de alquiler de bicicletas

Para poder promocionar un servicio como este es necesario analizar qué información necesita el usuario. Se desea saber dónde están las paradas y a cuanta distancia están del usuario. Se deberá poder mostrar esta información en forma de lista y de forma gráfica

mediante un mapa, de esta manera un usuario que conozca las estaciones no necesitará mirar el mapa, sino que solo necesitará revisar la lista para conocer la situación (bicicletas o huecos disponibles) de la estación. Se debe poder obtener la información de cuántas bicicletas hay en una estación porque si no hay ninguna libre, al usuario no le interesará seguramente ir a ella. De la misma manera, al usuario le interesa saber si hay huecos libres, ya que si ya tiene la bicicleta querrá saber en dónde podrá dejarla. Esta información de bicicletas disponibles/huecos libres es importante para dar un servicio completo, y por ejemplo en la aplicación Parkplona comentada anteriormente no se proporciona. Esta aplicación tampoco ofrece una interfaz en forma de lista que permita saber a cuánta distancia de cada estación se encuentra el usuario, funcionalidad que sí ofrece por ejemplo con los puntos de recarga de coches eléctricos. Si se desea un servicio más completo se podría monitorizar cada hueco de la estación, para saber si hay una bici aparcada o no, y mostrar de una manera más grafica la situación de la estación. Esta funcionalidad ya no depende solamente de la aplicación, debido a que hoy por hoy las estaciones no monitorizan esta información. Además, se debería implementar un sistema de favoritos para que los usuarios que usen el servicio habitualmente y lo hagan siempre usando las mismas estaciones accedan más rápidamente a la información que buscan.

Para finalizar, se va a analizar el sistema de registro en este servicio. Ahora mismo se necesita acudir al Ayuntamiento de Pamplona o a la oficina de Agenda 21 y presentar una fotocopia del DNI, una foto de carné y un número de cuenta bancaria del que se reservará una fianza de 150 euros. También se puede hacer online mediante la sede electrónica del ayuntamiento. Este sistema debería ser revisado de manera que fuera más accesible para todo el mundo. Ahora mismo el sistema está más enfocado para ciudadanos de Pamplona, los cuales pueden disfrutar de este servicio gratuitamente (ya que la fianza se devuelve al darse de baja del servicio) pero se podría promover para que los turistas visitaran la ciudad en bicicleta. Para ello sería necesario un cambio en el sistema de inscripción de forma

que se pudiera realizar mediante una aplicación (a poder ser la misma que ofrezca la información del servicio) de forma sencilla. Se podría plantear un pago por horas si el cliente introdujese la tarjeta de crédito al registrarse y una vez devuelta la bici se calcularía la tarifa a pagar por el servicio. De la misma manera que en la inscripción que hay en estos momentos se podría cobrar una fianza que luego se devolvería.

- Servicio de recarga de coches eléctricos

El ayuntamiento de la ciudad quiere promover el uso de vehículos eléctricos. Para ello ofrece ventajas y facilidades a los propietarios o usuarios de dichos vehículos. Como se ha comentado anteriormente, el estacionamiento en sitios regulados se prevé que sea gratuito para este tipo de vehículos. Además, una de las acciones del proyecto STARDUST es la instalación de 40 nuevos puntos de recarga para vehículos eléctricos [8], que se sumarán a los 4 disponibles en estos momentos. Al igual que para el servicio de bicicletas será necesario mostrar la información de la manera más gráfica posible. Para ello se mostrará la información relativa a los puntos de recarga en forma de lista y ordenada por cercanía al usuario. De estos puntos de recarga lo más interesante para un usuario es saber si están disponibles o no. Por ello, se deberá mostrar la disponibilidad e implementar algún sistema de reserva. Para este servicio el poder reservar el punto de recarga a unas horas concretas debería ser obligatorio. Esta funcionalidad facilitaría mucho la vida al usuario ayudándole a gestionar los tiempos de carga de su vehículo. Por el contrario, el implementar el sistema de reserva implica que los puntos de recarga estén preparados para esta funcionalidad y bloqueen su uso si hay alguna reserva efectuada. El usuario debería estar informado del coste del kilovatio por hora (kW/h) y de las posibles tarifas que se ofrezcan dependiendo de la hora a la que se haga la recarga (las horas nocturnas suelen ser más económicas). Adicionalmente a la vista en forma de lista se deberán mostrar todos los puntos de recarga en un mapa para ayudar al usuario a posicionar cada una. De la misma manera que en el servicio de bicicletas, se puede implementar un

sistema de favoritos para usuarios que usen recurrentemente los mismos puntos de recarga. De todo lo comentando anteriormente la aplicación Parkplona solo muestra una lista con los puntos y la distancia hasta ellos y el mapa con la situación de cada uno. Es decir, la información que a priori parece más interesante, que es la disponibilidad del punto de recarga, no se muestra.

En cuanto al pago por el servicio, se debería implementar algo parecido a lo citado en el análisis del servicio de bicicletas, es decir, un método de pago sencillo que te permita pagar a través del móvil con tarjeta, PayPal, NFC o algún sistema parecido. Se tendría que gestionar el tema de las reservas mediante alguna pequeña fianza para evitar que los usuarios hagan un mal uso del servicio de reserva.

- Cómo desplazarse por la ciudad

Uno de los principales objetivos del proyecto STARDUST, como se ha comentado con anterioridad, es gestionar adecuadamente la movilidad de la ciudad. Ya que la plataforma que se desarrollará contará con toda la información de movilidad de la ciudad sería beneficioso hacer uso de ella y ponerla al servicio de los ciudadanos. Si se tiene toda la información de las líneas de autobuses, estaciones de bicicletas y se puede calcular la distancia y tiempo que se tarda en viajar andando de un punto a otro de la ciudad, se puede ofrecer al usuario las mejores opciones para moverse por la ciudad de manera rápida y menos contaminante posible. Si el objetivo es reducir el uso del vehículo particular, al menos los que funcionan con motores de combustión, se le puede ofrecer al usuario rutas para recorrer la ciudad de manera que no tenga que hacer uso de este y que no tengan por qué ser siempre iguales. Al usuario se le pueden mostrar los tiempos y rutas para viajar de un punto a otro andando, en villavesa, en bicicleta o usando más de un medio, por ejemplo, parte en transporte urbano y parte en bicicleta. Esta información la debería calcular la plataforma y ser mostrada en la aplicación de manera gráfica, por ejemplo, mostrando rutas en un mapa. Esto además ayudaría a gestionar la movilidad de la ciudad con análisis de los desplazamientos efectuados por los ciudadanos. Si se sabe que

muchos usuarios se desplazan de una zona a otra con asiduidad y se ve que se puede mejorar ese desplazamiento cambiando alguna línea de transporte público o instalando nuevas estaciones de bicicletas se mejoraría la calidad de vida de los ciudadanos.

- Interacción con los ciudadanos

La colaboración ciudadana puede ser una herramienta importante a la hora de mejorar la gestión de la ciudad. Se debe impulsar y facilitar la comunicación entre los ciudadanos y su ayuntamiento para construir entre todos una ciudad mejor. Dar a los usuarios las herramientas adecuadas para ello es importante si se quiere impulsar esta comunicación. Por ello es necesario desarrollar un sistema de quejas, sugerencias o colaboración ciudadana, más conocido como un sistema de gestión de incidencias, que tenga su parte visible en la aplicación, digamos su front-end, y su gestión o back-end en la plataforma que se va a desarrollar. De esta manera se podrían categorizar y tratar adecuadamente cada uno de los tiques. Esto ayudaría a gestionar temas como mobiliario urbano defectuoso o roto siempre y cuando se consiguiera una comunidad activa de usuarios. Se podría premiar a estos usuarios mediante algún sistema de puntuación que recompensará a los más activos con pequeños premios como entradas de cine u otro tipo de actos celebrados en la ciudad. En cuanto a los tiques sería interesante ofrecer la posibilidad de mandar fotos, ubicar al usuario para localizar el problema en caso de ser una incidencia o incluso permitir mandar audios para evitar que la gente no participe por pereza de escribir el problema o queja.

- Información medioambiental

Al ser una aplicación sobre la ciudad es interesante mostrarle al usuario datos relativos al medioambiente de la ciudad. Se puede mostrar el tiempo meteorológico en tiempo real, que puede ser interesante para alguien que va a realizar un desplazamiento caminando o en bicicleta o simplemente por curiosidad. Tampoco estaría de más mostrar al usuario los datos de contaminación que recogiera la red de sensores que se montaría. Aunque a priori los

datos de contaminación son de mayor utilidad para el personal del ayuntamiento, para analizar zonas con más contaminación y tomar medidas, hay datos como la temperatura o humedad que sí podrían ser interesantes para un usuario de la aplicación. Si la red de sensores fuera extensa se podría mostrar al ciudadano los valores más cercanos a su posición, lo que haría más precisa la información mostrada y mejoraría el servicio ofrecido. Además, se podría incluir información sobre los niveles de polinización en la ciudad para las personas alérgicas al polen.

A continuación, se van a analizar los objetivos de la red de sensores que se quiere desarrollar y lo necesario para poner la red en marcha.

- Cuanto más se analice mejor

  Cada nodo de la red deberá contar con la mayor cantidad de sensores posibles. Se deberán seleccionar sensores que recojan los datos de contaminación más relevantes, es decir, nos interesa más saber acerca de los niveles de contaminación de un gas habitual que de uno cuyos niveles suelan ser prácticamente nulos. Este hecho también hará que los resultados a analizar sean más interesantes y permitan sacar conclusiones al comparar datos obtenidos en localizaciones diferentes.

- La conexión de la red debe ser estable

  Al ser una comunicación a priori inalámbrica, aunque se podría plantear una red cableada siempre y cuando se entiendan y analicen los beneficios e inconvenientes de ésta, es fundamental asegurarse de que la comunicación entre los nodos y el servidor sea viable. Se deberá asegurar la comunicación entre ambos extremos, ya sea por distancia o por obstáculos entre ambos. Se deberá hacer uso de repetidores o cambiar la tecnología de comunicación para asegurar el buen funcionamiento de la red.

- Un sistema que gestione los datos

La red debe contar con un sistema que gestione los datos como puede ser un servidor. Este servidor deberá estar conectado a la red para poder recibir los paquetes enviados por los nodos y contar con un sistema de gestión de base de datos para almacenar, actualizar u obtener los datos obtenidos. Este servidor deberá estar conectado a internet o a alguna red para facilitar la extracción de los datos. Este servidor en un futuro debería ser sustituido por la plataforma que se va a desarrollar, de manera que todos los datos se centralicen en una única plataforma. Otra opción sería que el servidor desarrollado se comunicara con la plataforma para enviarle los datos de forma periódica, aunque el hecho de añadir más pasos en la fase de comunicación, nodos-plataforma frente a nodos-servidor-plataforma, también hace que pueda haber más puntos donde la comunicación falle.

2.2 Requisitos

Una vez analizados los objetivos, tanto de la aplicación como de la red de sensores, es el momento de definir los requisitos funcionales y no funcionales fijados para el proyecto. Estos requisitos han sido fijados a partir de los objetivos marcados anteriormente.

2.2.1 Requisitos funcionales

I. Interfaz de usuario que siga la regla de los tres clics.
II. Pantalla de inicio que permita ver todos los servicios que ofrece la aplicación.
III. Base de datos que permita guardar información de las estaciones de bicicletas, puntos de recarga y favoritos.
IV. Pantalla de servicio de alquiler de bicicletas con tres vistas: vista de lista, vista de mapa y vista de favoritos.
V. Pantalla de servicio de alquiler de bicicletas en forma de lista que ofrezca información de bicicletas y huecos disponibles.
VI. Pantalla de servicio de alquiler de bicicletas en forma de lista que permita añadir una estación a favoritos.
VII. Pantalla de servicio de alquiler de bicicletas en forma de lista que ofrezca información de distancia a la estación.

VIII. Pantalla de servicio de alquiler de bicicletas en forma de mapa que ofrezca al usuario cómo llegar a la estación.
IX. Pantalla de servicio de recarga de coches eléctricos con tres vistas: vista de lista, vista de mapa y vista de favoritos.
X. Pantalla de servicio de recarga de coches eléctricos en forma de lista que ofrezca información de disponibilidad.
XI. Pantalla de servicio de recarga de coches eléctricos en forma de lista que permita añadir un punto de recarga a favoritos.
XII. Pantalla de servicio de recarga de coches eléctricos en forma de lista que ofrezca información de distancia al punto de recarga.
XIII. Pantalla de servicio de recarga de coches eléctricos en forma de mapa que ofrezca al usuario cómo llegar al punto de recarga.
XIV. Pantalla de cómo desplazarse de un punto a otro de la ciudad en forma de mapa.
XV. Pantalla de gestión de incidencias que permita generar un tique para el sistema de incidencias.
XVI. Pantalla de información medioambiental que muestre en forma de gráfico los datos obtenidos por la red de sensores
XVII. Pantalla de información medioambiental que permita seleccionar los valores a visualizar.
XVIII. Conexión entre la red de sensores y la plataforma estable e inalámbrica.
XIX. Plataforma con un gestor de base de datos para almacenar los datos.
XX. Plataforma con un servicio web para acceder a los datos desde la aplicación.

2.2.2 Requisitos no funcionales

I. Compatible con todas las versiones Android superiores a la 4.0.
II. Interfaz de usuario que siga las pautas del Material Design.
III. La aplicación debe estar operativa para antes de junio de 2018.

2.3 Estructura del proyecto

Una vez fijados los requisitos que tendrá el proyecto se analizará la estructura que este tendrá. El proyecto estará estructurado en tres partes: la aplicación Android, la plataforma que hará de intermediario entre la red de sensores y la aplicación y, por último, la red de sensores. El usuario tan solo interactuará con la aplicación y la aplicación se encargará de interactuar con la plataforma cuando sea necesario. La comunicación entre la red de sensores y la plataforma será continuada, ya que cada cierto tiempo los nodos que formen la red recogerán los datos obtenidos mediante los sensores y se los mandarán a la plataforma. En la Ilustración 9 se puede observar cómo se estructurará el proyecto y la interacción entre sus diferentes componentes.

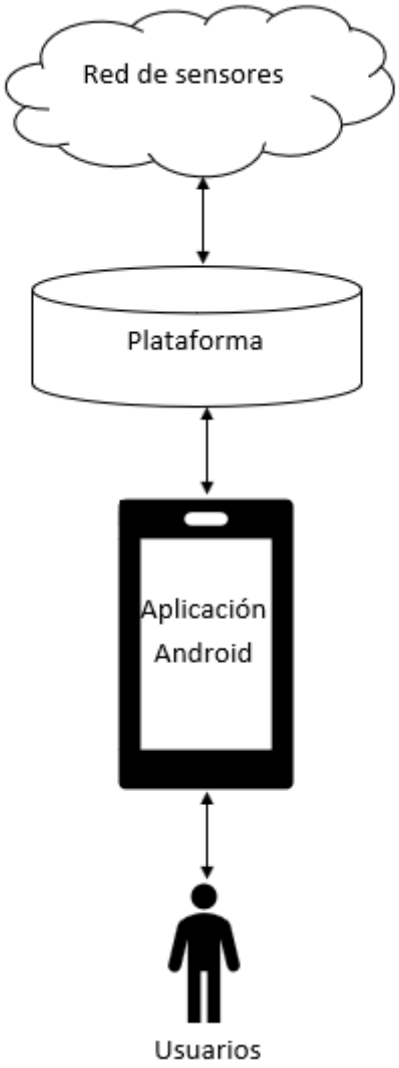

Ilustración 9. Estructura del proyecto

3. Diseño

En este capítulo se describirá el diseño planteado para el desarrollo del proyecto basándose en los análisis realizados previamente. Se describirá la interfaz planteada para la aplicación Android, así como la estructura de la red de sensores. También se analizará la comunicación entre la aplicación y la red de sensores. En el diseño se tendrán en cuenta todos los requisitos, sobre todo los funcionales, fijados para el proyecto. Una vez explicado el diseño de la aplicación y de la red se procederá a analizar el entorno de desarrollo y las herramientas que se usarán.

3.1 Mockups de la aplicación

Lo primero que se debe hacer a la hora de desarrollar una aplicación es diseñar la interfaz de la aplicación, o al menos tener una idea de cómo se quiere que sea ésta. A estos diseños iniciales se les conoce como mockups y en las empresas por lo general suele haber una o varias personas encargadas del diseño gráfico. Los diseños tienen en cuenta la experiencia de usuario que marca el Material Design citado anteriormente. Sin embargo, no todos los mockups tienen por qué corresponderse a las pantallas finales. Algunas pantallas podrán sufrir cambios e incluso se pueden añadir nuevas pantallas o eliminar alguna. Estos diseños se suelen hacer con herramientas de diseño como Adobe Photoshop o Illustrator, aunque en este caso han sido realizadas con una aplicación web sencilla llamada Moqups [14]. El resultado no es sorprendente, pero cumple su objetivo, que es dar una idea de la interfaz de la aplicación. Algunas pantallas en un principio no fueron diseñadas y se dejaron para el momento del desarrollo como se explicará en el próximo capítulo. Entre los mockups realizados se encuentran las siguientes pantallas:

- Pantalla principal o de inicio (Ilustración 10).
- Pantalla de alquiler de bicicletas (Ilustraciones 11, 12, 13).
- Pantalla de estaciones de recarga (Ilustraciones 14, 15, 16).
- Pantalla de quejas e incidencias (Ilustración 17).

A continuación, se muestran las pantallas diseñadas.

Ilustración 10. Mockup pantalla inicial

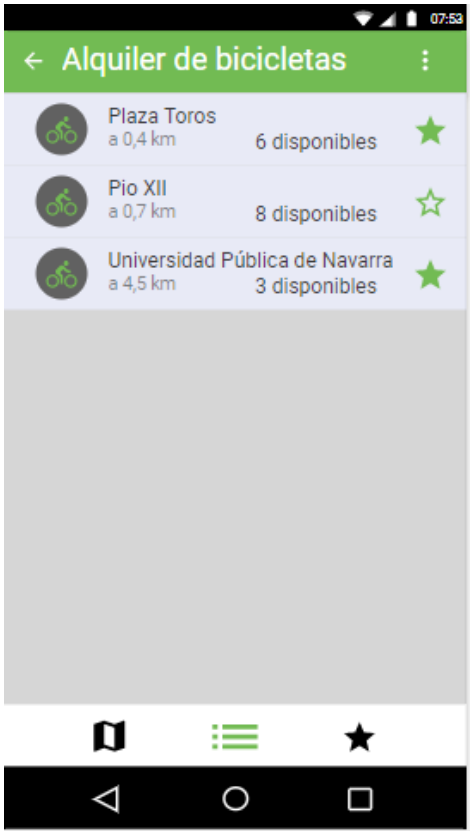

Ilustración 11. Mockup lista de bicicletas

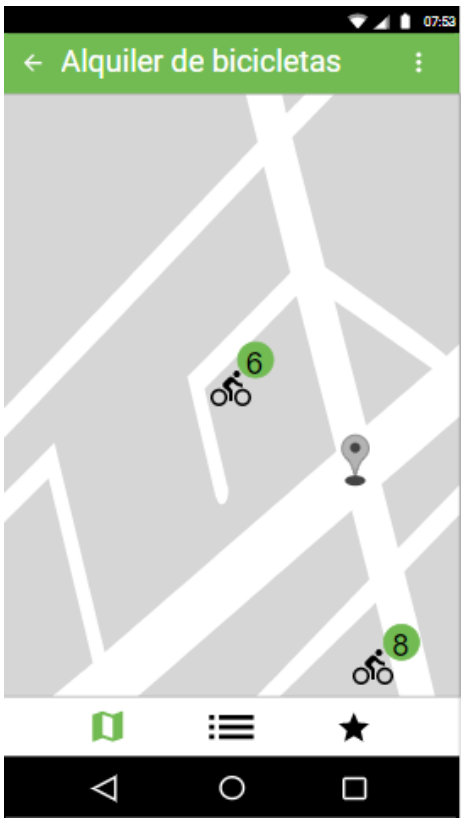

Ilustración 12. Mockup mapa de bicicletas

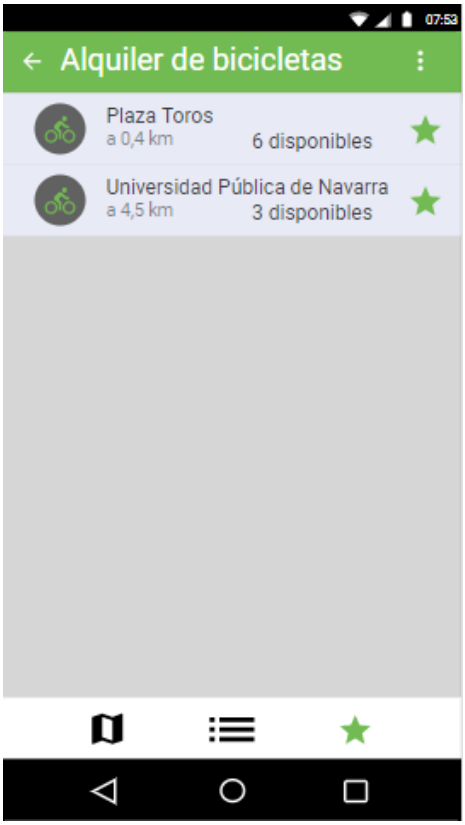

Ilustración 13. Mockup favoritos de bicicletas

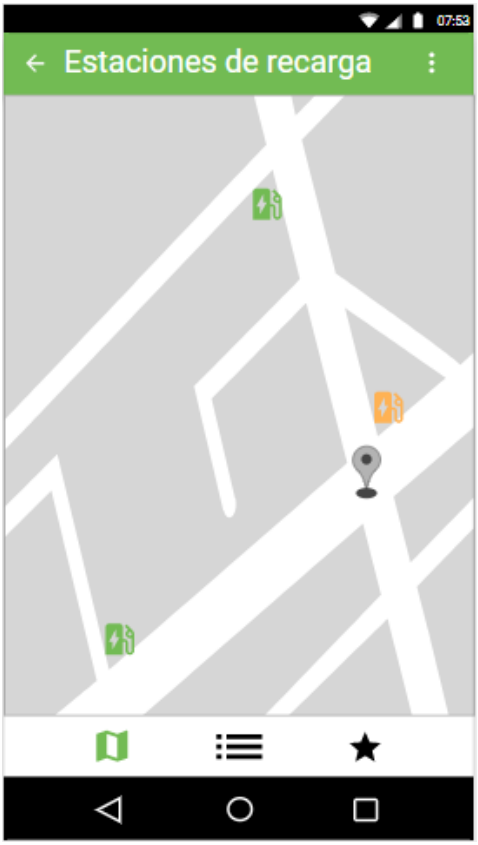

Ilustración 14. Mockup mapa de estaciones

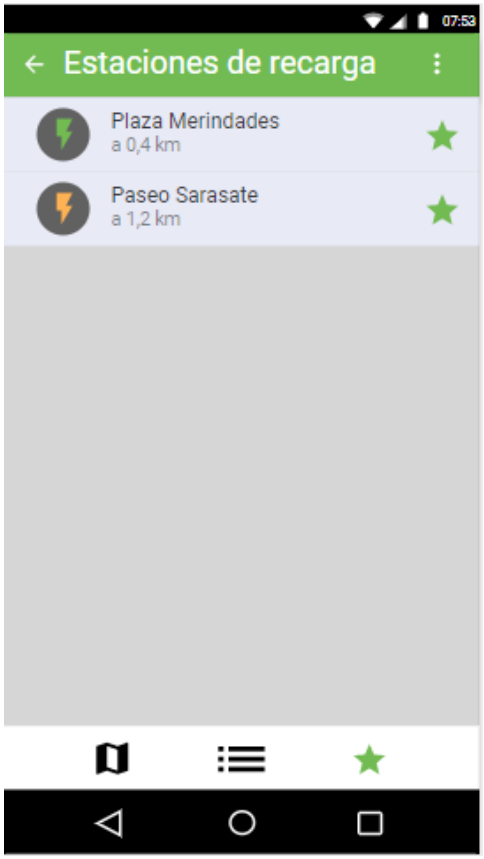

Ilustración 15. Mockup favoritos de estaciones

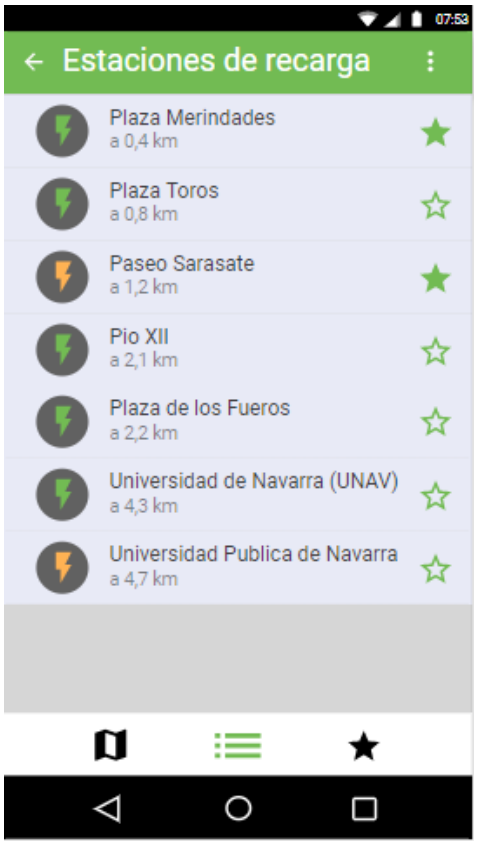

Ilustración 16. Mockup lista de estaciones

Ilustración 17. Mockup pantalla de incidencias

3.2 Estructura de la red de sensores

La red de sensores está formada por nodos conocidos como motas. En nuestro caso por Waspmotes de la empresa Libelium [15] (Ilustración 18). Cada uno de estos nodos es una placa que ha tenido que ser programada para recoger información y mandarla al servidor que se encargará de recoger y almacenar los datos. Estas placas cuentan con algunas características que se comentarán a continuación:

- **Batería externa**. Cuentan con una batería externa que alimenta la placa cuando ésta no está conectada por micro USB.
- **Antenas**. Cuentan con módulos para conectar antenas a la placa. Estas antenas pueden ser de diferentes tipos como antenas WIFI o ZigBee entre otras.
- **Placas de sensores**. Cuentan con un módulo para conectar una placa de sensores. Entre los sensores disponibles se encuentran sensores de parking, sensores de radiación, sensores para agricultura, sensores de gases, etc.
- **Tarjeta SD**. Cuentan con una ranura SD para almacenar la información en ella en vez de mandarla mediante comunicación inalámbrica.

Ilustración 18. Componentes placa Waspmote

Para la red que nos ocupa se montarán placas con baterías externas, placas de sensores de gases y antenas ZigBee. En cada placa de sensores se montarán seis sensores:

- Sensor de temperatura.
- Sensor de humedad.
- Sensor de monóxido de carbono (CO).
- Sensor de dióxido de carbono (CO_2).
- Sensor de contaminantes de aire, entre los que se encuentran el metilpropano (C_4H_{10}), el etanol (CH_3CH_2OH), el hidrogeno (H_2), el monóxido de carbono (CO) y el metano (CH_4).
- Sensor de compuestos orgánicos volátiles que detecta gases que contienen carbono.

Ilustración 19. Placa de sensores de Waspmote

Para la red de comunicación se utilizará el protocolo ZigBee. Este protocolo de comunicación inalámbrica está basado en el estándar IEEE 802.15.4. En concreto, se usarán los módulos XBee que usan la banda de frecuencia de 2.4 GHz y emplean 16 canales. Estos módulos están pensados para domótica y lo más interesante es su bajo consumo. En cuanto a la topología de la red se usará una topología en forma de estrella como se muestra en la Ilustración 20.

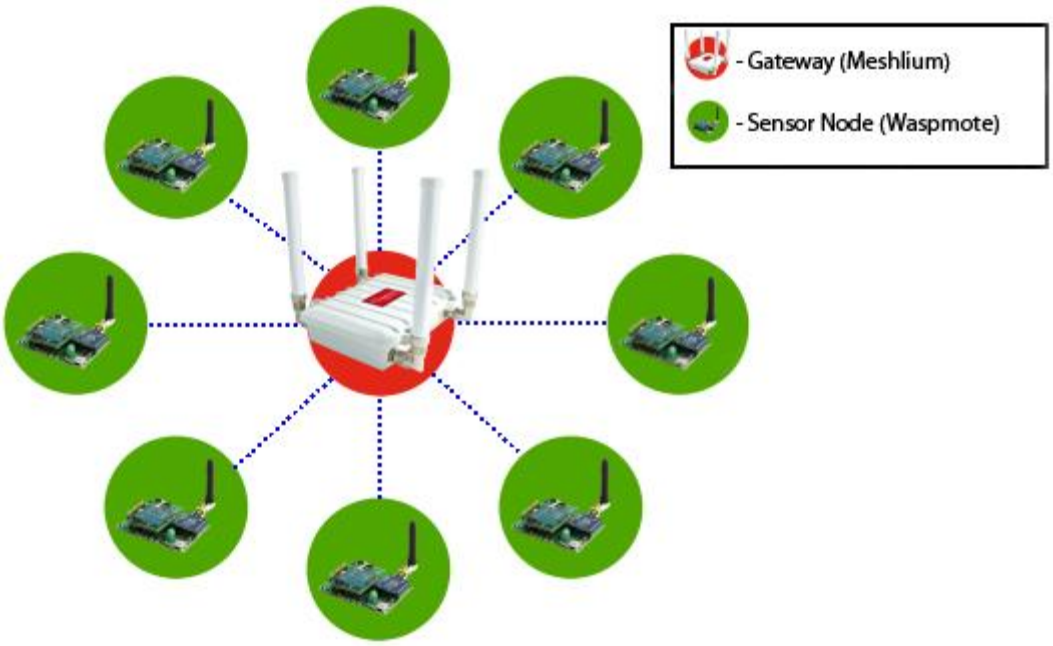

Ilustración 20. Topología de red en estrella

En esta topología todos los nodos están conectados a un punto central. Se trata por lo tanto de una red centralizada en la que los nodos tienen una comunicación de difusión única ("unicast"), en la cual el envío de información se realiza de un único emisor a un solo receptor. Además, esta comunicación es unidireccional, ya que los nodos son los que envían la información al servidor y nunca al contrario.

En el centro de la red se encuentra un servidor denominado Meshlium [16]. Este aparato es un servidor de los propios desarrolladores de las placas Waspmote (Libelium). Viene configurado para trabajar con las motas de forma predeterminada, de manera que, si se programan correctamente las placas para que las tramas se envíen a este servidor, el propio servidor se encarga de tratar las tramas recibidas y almacenar la información en la base de datos que tiene instalada. Al sistema Meshlium (Ilustración 21) se le conectan las antenas ZigBee mediante las que recibirá la información.

Ilustración 21. Meshlium

El servidor lleva instalado un sistema operativo Linux, en concreto Debian, y una base de datos MySQL. Es posible conectarse a él mediante SSH o gracias al interfaz web que incorpora. Por defecto, Meshlium viene con un interfaz Wifi de 2.4 GHz preparada para funcionar como punto de acceso (AP). Esto significa que crea una red que se puede acceder desde cualquier ordenador o dispositivo y permite configurar cómodamente los aspectos más comunes mediante un interfaz web. También cuenta con un servicio DHCP para que cuando un usuario se conecte a través de una conexión Wifi se le asigne automáticamente una IP privada en el rango 10.10.10.10 - 10.10.10.250. Para temas de configuración más complicados es necesario realizarlo mediante SSH, aunque el propio fabricante no lo recomienda dado que si le pasa algo al aparato no se hacen responsables. Mediante esta interfaz se pueden configurar los parámetros de la red de sensores como el Network ID o el canal. Además, cuenta con un capturador de paquetes que permite ver las tramas recibidas en tiempo real y ver el contenido de la base de datos. La comunicación en general quedaría tal y como se muestra en la Ilustración 22.

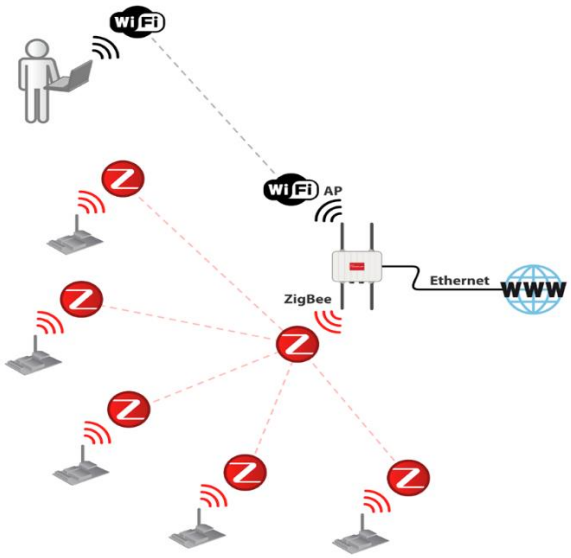

Ilustración 22. Comunicación de la red

En la Ilustración 22 se aprecia la red en estrella que forman los nodos con el punto de entrada (“Gateway”), que en este caso es el Meshlium, la comunicación entre el programador y el servidor, mediante el punto de acceso Wifi que éste crea para poder configurarlo, y el acceso desde internet para poder acceder a los datos almacenados. En la Ilustración 23 se aprecian los nodos que forman la red con todos sus componentes (batería, antena y sensores).

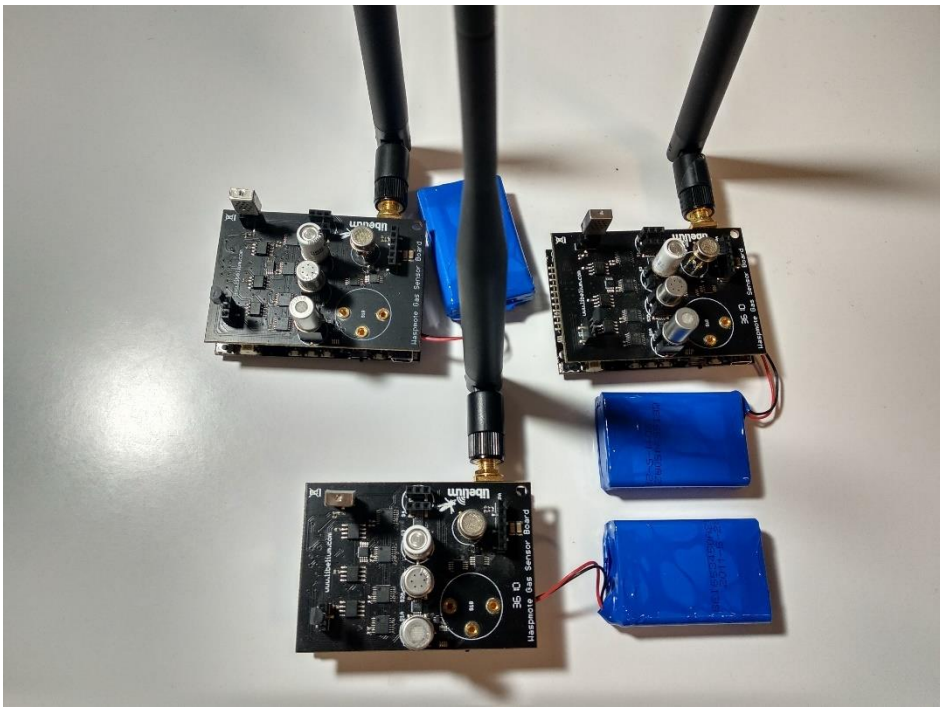

Ilustración 23. Nodos Waspmote de la red

Cambio del servidor

El diseño planteado anteriormente sufrió un cambio debido a errores con el servidor Meshlium. Tras algunas pruebas iniciales con tramas sin datos de los sensores se observó que el servidor se colapsaba llegado un momento, seguramente debido a que no era capaz de procesar todas las tramas que le llegaban. Además, el acceso a los datos desde la aplicación no iba a ser directa, ya que Meshlium no ofrece un servicio web para poder acceder a ellos mediante una petición. Por estas razones se cambió el planteamiento del servidor. Teniendo en cuenta los objetivos y requisitos que el servidor debía satisfacer analizadas en el capítulo anterior, se planteó una arquitectura LAMP. LAMP es el acrónimo de Linux + Apache + MySQL + PHP. Esta infraestructura es comúnmente usada para servidores web.

Utiliza Linux como sistema operativo, el cual es software libre y de código abierto. Este sistema operativo será la base para todo lo que se instalará para poner en marcha el servicio.

Apache es un servidor web que utiliza el protocolo HTTP y que al igual que Linux es de código abierto. Apache procesa la información del lado del servidor realizando conexiones con el cliente y generando respuestas a las peticiones de éste. Esto permitirá ofrecer la información a la aplicación para que la información almacenada pueda ser mostrada y a su vez, también permitirá guardar la información que recibirá de la red de sensores en la base de datos con la que contará este servidor.

El sistema gestor de base de datos (SGBD) que se usará es MySQL. Es la base de datos de código abierto más popular y usada para entornos de desarrollo web [17]. Ésta se encargará de almacenar los datos obtenidos y de facilitar la información solicitada por el servicio web.

Por último, PHP será el lenguaje de programación que se usará para desarrollar el servicio web que se ejecutará en el servidor Apache y que se comunicará con la base de datos cuando sea oportuno.

El hardware que se usará para montar todo esto será una Raspberry Pi 3 Model B. Esta placa [18] (Ilustración 24) cuenta con una CPU de cuatro

núcleos con una velocidad de 1.2 GHz, 1 GB de memoria RAM, conectividad Wifi y Bluetooth, una entrada HDMI, 4 puertos USB, un puerto RJ-45 y un puerto microSD para poder cargar el sistema operativo. Estas características se consideran suficientes para lo que se pretende que haga para este trabajo. Hay que tener en cuenta que en un futuro el servidor que realice este trabajo será la plataforma Smart City que se desarrollará. Para poder acometer su objetivo, a la placa se le instalará el sistema operativo Raspbian Pixel (Linux) que ofrecerá un interfaz gráfico visualizable a través del HDMI de la placa. Sobre este se instalará el servidor Apache y después la base de datos MySQL y PHP que completarán la infraestructura LAMP.

Ilustración 24. Raspberry Pi 3 Model B

3.3 Herramientas y entorno de desarrollo

En este apartado se analizarán las herramientas y demás componentes que forman el entorno de desarrollo que se usará para la implementación del trabajo. Aquí se hablará tanto de las herramientas que se usarán para el desarrollo de la aplicación en Android, como las usadas para la red de sensores y el sistema de recogida de datos.

3.3.1 Android

Android [9] es un sistema operativo lanzado por Google en el año 2008. Está basado en el sistema operativo Linux y fue diseñado para dispositivos con pantallas táctiles, que por aquel entonces estaban en auge. El sistema operativo está compuesto por millones de líneas de código en XML, C, Java y C++. Android en lugar de utilizar la máquina virtual Java (VM) utiliza su propia máquina virtual llamada Dalvik que fue diseñada para asegurar una ejecución multitarea eficiente en el

dispositivo. Mediante el uso de esta máquina virtual para la ejecución de aplicaciones, los desarrolladores no tienen que desarrollar nada de la capa de hardware, preocupándose solo del nivel de aplicación.

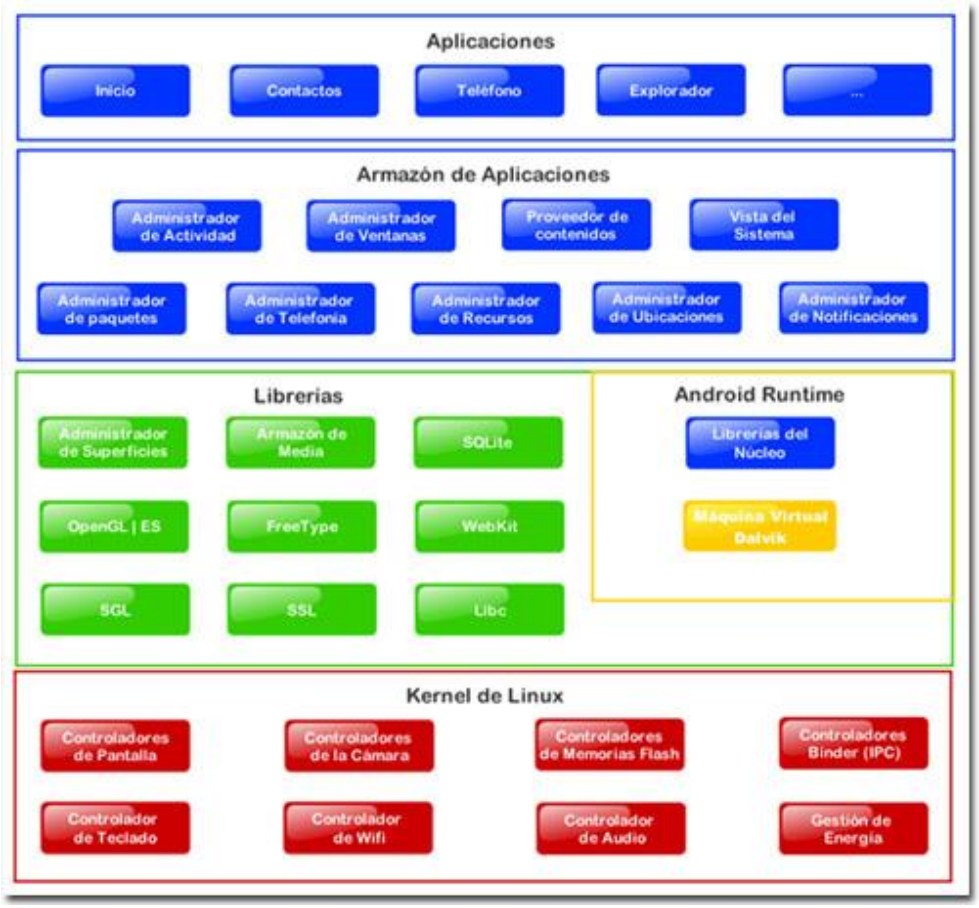

Ilustración 25. Arquitectura de Android

Android utiliza la base de datos SQLite para el almacenamiento de datos y cuenta con un catálogo de aplicaciones denominada Google Play que facilita la descarga e instalación de aplicaciones en el sistema. Actualmente el sistema operativo se encuentra en la versión 8.1, más conocida como Oreo. La cuota de mercado en España de este sistema supera el 86% hoy en día según un estudio de Kantar Worldpanel [10]. Como se puede ver en Ilustración 26 su evolución en la cuota de mercado desde enero de 2012, cuando la última versión era la 4.0 Ice Cream Sandwich, hasta marzo de 2018 ha sido bastante notoria. Como anécdota hay que comentar que todas las versiones de Android han tenido un nombre relacionado con comida, generalmente dulces o chucherías.

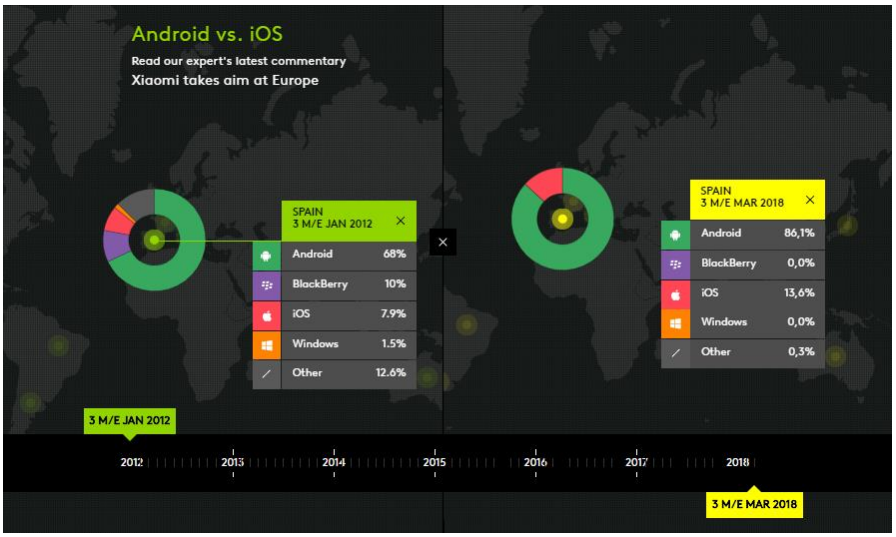

Ilustración 26. Cuota de mercado de Android en España [10]

3.3.2 Android Studio IDE

Android Studio es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones en Android. Un IDE es básicamente una aplicación que facilita el desarrollo de software a los programadores. Estas aplicaciones constan de un editor de textos, un compilador, un depurador de código, herramientas de construcción automáticas y por lo general, de un autocompletado de código inteligente. Al ser programas que facilitan en gran medida el desarrollo, gracias a las herramientas y funcionalidades que ofrecen, casi todos los desarrolladores hacen uso de ellos. En el caso de Android, la mayoría de los programadores optan por el software mencionado para facilitar el desarrollo de aplicaciones para este sistema. Este IDE ha sido desarrollado por Google basándose en IntelliJ IDEA de JetBrains, empresa que también desarrolló el lenguaje de programación Kotlin del que se hablará más adelante. A continuación, se comentarán las diferentes zonas que forman la interfaz de la aplicación y las funcionalidades que ofrecen tal y como se puede observar en la Ilustración 27.

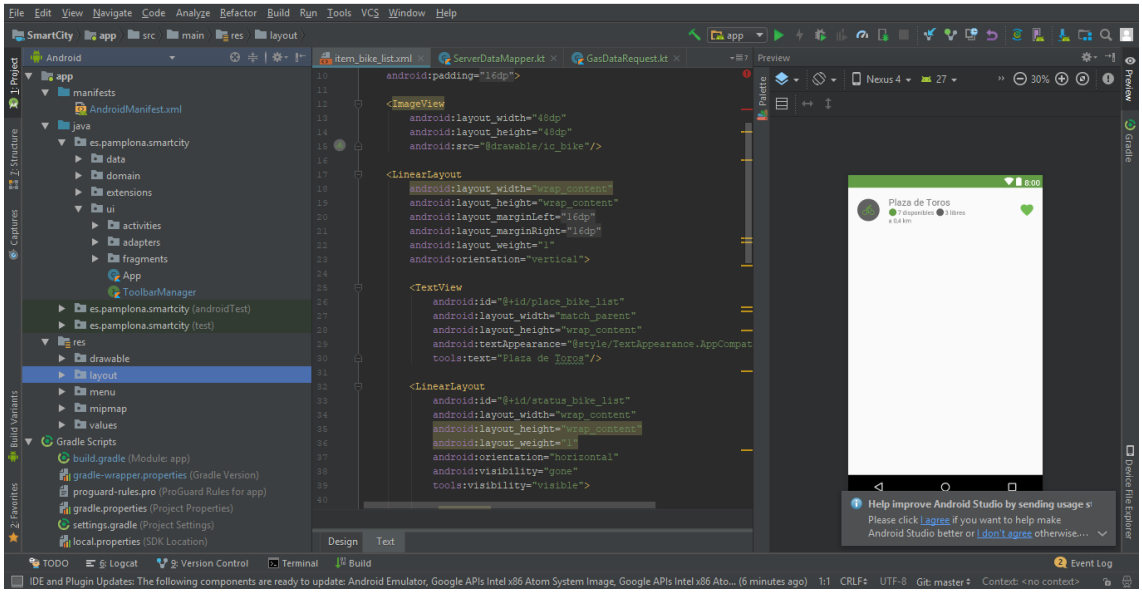

Ilustración 27. Interfaz de Android Studio

- Barra de herramientas superior

  Esta barra contiene las opciones típicas de configuración de un programa y está situada en la parte de arriba de la interfaz. Ofrece de izquierda a derecha:

  - Opciones de fichero (guardar y cargar proyectos, etc.).
  - Ajustes de edición (copiar, pegar, etc.).
  - Opciones de visualización.
  - Opciones de navegación.
  - Ajustes, análisis y refactorización del código.
  - Opciones de construcción y ejecución del proyecto.
  - Herramientas de programación.
  - Ajustes de control de versiones.
  - Pestaña de ayuda.

- Barra de herramientas de proyecto

  Ésta ofrece opciones relacionadas con el proyecto, como por ejemplo la ejecución o depuración de la aplicación. Está situada debajo de la barra de herramientas superior.

- Explorador de archivos

  Muestra todos los archivos y carpetas que forman el proyecto. Está situada en la parte izquierda de la interfaz.

- Editor de texto
  Se sitúa en la parte central del IDE y muestra el fichero que se encuentre abierto en ese momento. Permite editar el código y ofrece un autocompletado inteligente que facilita la redacción del código.
- Editor gráfico
  Permite ver la vista previa de las pantallas desarrolladas y modificar los elementos que la componen mediante desplazamientos de ratón.
- Ventanas de ejecución y depuración
  Muestra datos de la aplicación al ser ejecutada o depurada, lo que permite comprobar su correcto funcionamiento.

Por último, hay que comentar que la Ilustración 27 corresponde a la versión 3.1.2 del programa.

3.3.3 Kotlin

Kotlin [11] es un lenguaje de programación desarrollado por la empresa JetBrains. Fue lanzado en el año 2012, pero fue en el año 2017 cuando Google anuncio que Kotlin sería un lenguaje oficial de desarrollo de aplicaciones para Android, uniéndose así a Java. De esta manera Google empezó a dar soporte a este lenguaje, asegurándose de que su IDE y las librerías funcionaran sin problemas con él. Desde la versión 3.0 de Android Studio es posible desarrollar en Kotlin sin hacer uso de plugins que en las versiones anteriores eran necesarios. Kotlin fue específicamente creado para el desarrollo de aplicaciones en Android y su objetivo era mejorar a Java en lo que a estos desarrollos se refiere. Kotlin ofrece las ventajas de un lenguaje moderno, que más adelante se comentarán, junto con las ventajas de ser un lenguaje basado en la máquina virtual de Java (JVM). Es un lenguaje orientado a objetos y completamente interoperable con Java, hecho que facilita la migración de Java a Kotlin. A continuación, se analizarán las ventajas de Kotlin respecto a Java, ya que es la comparación natural para alguien que se está planteando desarrollar una aplicación Android.

- **Es más expresivo**, es decir, puedes desarrollar mucho más con menos código. Desarrollar el mismo código en Java nos llevará más líneas que en Kotlin.
- **Es más seguro**. Esto se debe a que evita errores por nulos, ya que trata las posibles situaciones en las que se pueden dar nulos en tiempo de compilación. De esta manera evita nulos en tiempo de ejecución obligando al programador a realizar comprobaciones antes de usar una variable que pueda ser nula. Para que un objeto pueda ser nulo se deberá indicar, porque por defecto todos los objetos en Kotlin no pueden ser nulos. Esto ahorra tiempo a los programadores solucionando errores por punteros nulos (Null Pointer Exception), que en muchas ocasiones obligan a depurar el código.
- **Es funcional**. Es un lenguaje orientado a objetos y no un lenguaje puramente funcional, pero al igual que otros lenguajes modernos usa conceptos de la programación funcional como las expresiones lambda de las que se hablará más adelante.
- **Hace uso de funciones de extensión**. Esto significa que se puede extender cualquier clase, aunque no se tenga acceso al código fuente.
- **Es altamente interoperable**. Se pueden seguir usando todas las librerías y códigos desarrollados en Java gracias a la excelente interoperabilidad entre ambos lenguajes. Esto facilita la migración de Java a Kotlin para proyectos que quieren ser actualizados.

Vistas las diferencias que tiene con Java se van a analizar las principales características de este lenguaje.

- **Expresividad**. Con Kotlin es mucho más fácil evitar la repetición debido a que los patrones más comunes están cubiertos por defecto por el lenguaje. En la Ilustración 28 se puede observar la diferencia entre crear una clase de datos en Java con sus “getters” y “setters” y la misma clase en Kotlin. A la izquierda de la imagen está la clase en Java y a la derecha su equivalente en Kotlin. Como vemos, hay una diferencia notable en lo que a líneas de código se

refiere, y esto teniendo en cuenta que la clase solo cuenta con dos propiedades, si no fuera así la diferencia sería aún más notoria.

```java
public class Persona {
    private long id;
    private String nombre;

    public Persona(long id, String nombre) {
        this.id = id;
        this.nombre = nombre;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public long getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}
```

```kotlin
data class Persona(var id: Long, var nombre: String)
```

Ilustración 28. Diferencia entre Kotlin y Java

- **Seguridad con nulos**. Kotlin al igual que muchos lenguajes modernos es seguro con nulos debido a que el tipo define si un objeto puede ser nulo mediante el operador de llamada segura (escrito mediante "?"). A continuación, se muestran unos ejemplos de su uso y fallos de compilación.

```kotlin
var personaNoNula: Persona = null    // Error de compilación
var persona: Persona? = null         // Correcto

persona.print()                      // Error de compilación
persona?.print()                     // Correcto
if(artist != null) {
    artist.print()                   // Correcto
}

```

Ilustración 29. Nulos en Kotlin

La expresión de la primera línea daría un error de compilación porque el tipo Persona no se ha especificado como nullable, cosa que si se ha hecho en la expresión de la segunda línea. La expresión de la cuarta línea daría un error en tiempo de compilación por ser una variable que puede ser nula y no comprobarlo. Por último, la expresión de la quinta línea y el if de la sexta son equivalentes y ambas son correctas ya que primero se comprueba que la variable no sea nula y luego se imprime.

- **Funciones de extensión**. Se pueden añadir nuevas funciones a cualquier clase sin tener que usar las típicas clases de utilidades, conocidas como “utility classes”.
- **Funciones a la par de las clases**. Esto significa que las funciones están al mismo nivel que las clases, es decir, no es necesario crear una clase para definir una función. Esto permite que una función pueda recibir otra función como argumento o que devuelva una función.
- **Soporte funcional (Lambdas)**. Las expresiones lambda son útiles para definir funciones anónimas. Permiten no tener que definir funciones abstractas en clases y luego implementarlas. Además, como ya se ha comentado, las funciones están al mismo nivel que las clases por lo que una expresión lambda, al tener el comportamiento de una función anónima, también puede ser pasada como argumento de otra función, ser devuelta por una función, ser guardada en una variable, etc. En la Ilustración 30 se muestra cómo el uso de lambdas puede simplificar el código respecto a Java (a la izquierda). Básicamente se le pasa una función lambda que define el comportamiento del clic como parámetro a la función setOnClickListener.

```java
public interface OnClickListener {
    void onClick(View v);
}

view.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        Toast.makeText(v.getContext(), "Click", Toast.LENGTH_SHORT).show();
    }
});
```

```kotlin
view.setOnClickListener { toast("Click") }
```

Ilustración 30. Uso de lambdas en Kotlin

- **Operaciones funcionales en colecciones**. Esto permite trabajar con colecciones (listas, mapas, etc.) de una manera rápida. Esto ahorra código y Kotlin ofrece muchas funciones para operaciones de agregación, de filtrado, de mapeado, de elemento, de ordenación, etc. En vez de tener que recorrer la colección para, por ejemplo, aplicarle un filtro, bastará con aplicar la función de filtro y especificar el filtro a usar. En la Ilustración 31 se muestra un ejemplo de cómo obtener todos los valores pares de una lista aplicando un filtro.

```
1  val list = listOf(1,2,3,4,5,6)
2  list.filter { it % 2 == 0 }
```

Ilustración 31. Filtrado de una colección en Kotlin

Además de las caracterizas hasta ahora nombradas, Kotlin cuenta con otras muchas pequeñas ventajas que hacen que la programación sea más cómoda y no van a ser nombradas aquí.

Para finalizar, se va a comentar un ejemplo de una aplicación conocida y con más de 100.000 descargas en Play Store. Se trata de Basecamp 3, una aplicación para gestión de proyectos y comunicación de equipo. En un post [12] cuentan cómo fue migrar la aplicación desarrollada en Java a Kotlin por completo, con el 100% del código escrito en este lenguaje. En el citado post comentan cómo la felicidad de los programadores mejoró, así como la calidad del trabajo y la velocidad. También se comenta que el conversor automático de Java a Kotlin que se incorpora en Android Studio puede ayudar mucho a aprender cómo funciona Kotlin y sus diferencias con Java. Aun así, no recomiendan usarlo de una forma desmesurada, ya que hay expresiones que el conversor tal vez no las convierta de manera eficiente a Kotlin.

3.3.4 Waspmote IDE

Se trata de un IDE para el desarrollo de las placas Waspmote que se usarán como nodos de la red de sensores. Esta aplicación permite desarrollar el código en C++ y subir el programa a las placas. Este IDE es mucho más sencillo que otros como por ejemplo Android Studio. No cuenta con autocompletado de código ni con depurador, pero tiene lo básico que es un editor de textos, el compilador y una consola. Las zonas que forman la interfaz mostrada en la Ilustración 32 son las siguientes:

- Barra de herramientas superior
  Muestra las opciones del IDE en cuestión de ficheros, edición, ejecución, herramientas y ayuda.
- Barra de opciones de programa
  Permite compilar el programar, parar la compilación, crear nuevos programas, abrir un programa, guardar el programa, subir el programa a la placa y monitorizar el puerto serie.
- Editor de texto

Permite editar los programas como cualquier editor simple.

- Terminal

  Muestra los resultados de la compilación y la subida de los programas a las placas.

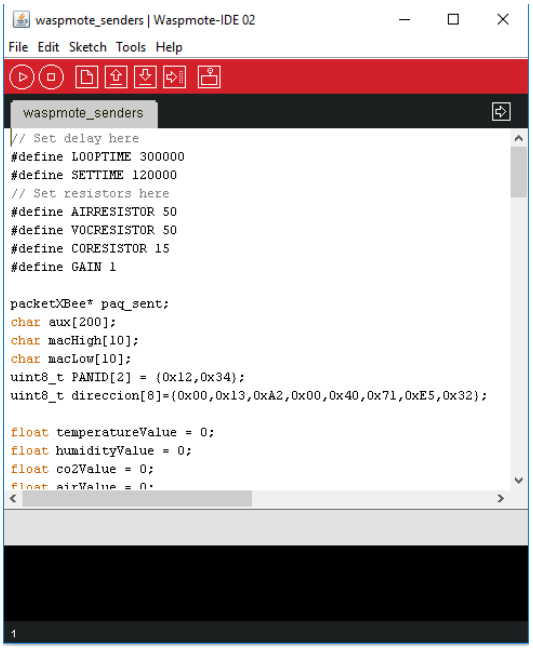

Ilustración 32. Waspmote IDE

3.3.5 Raspbian

Raspbian [13] es un sistema operativo desarrollado para la Raspberry Pi. Al ser específicamente diseñado para este dispositivo está bien optimizado para el hardware con el que cuenta la placa. Raspbian está basado en una distribución del sistema GNU/Linux, en Debian para ser más específicos. Cuenta con dos versiones: Raspbian Pixel que cuenta con un entorno gráfico y Raspbian Lite que solo cuenta con una consola. Para este proyecto se usará la versión Pixel para hacer más cómoda su configuración y programación. Será necesario instalar todas las herramientas necesarias para el proyecto ya que viene con aplicaciones bastante básicas como Libre Office, un navegador y alguna herramienta de programación no muy potente.

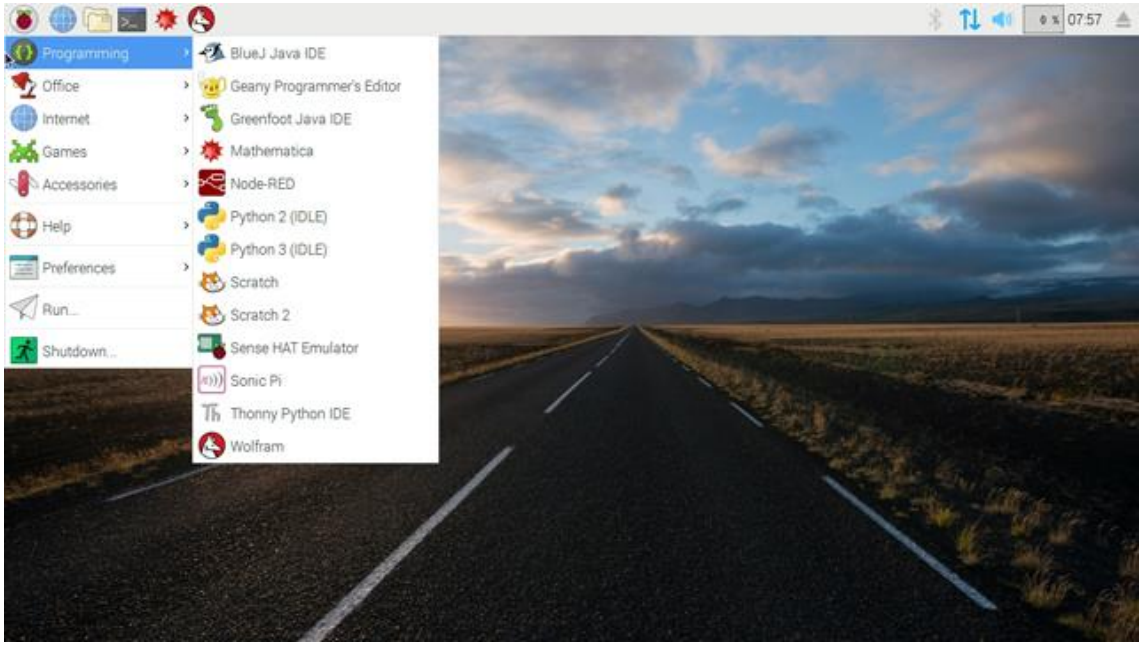

Ilustración 33. Escritorio de Raspbian Pixel

4. Desarrollo de la solución

En este capítulo se describirá técnicamente la solución implementada. Se explicará el desarrollo de la aplicación Android, así como todos los elementos que forman parte de ella, pasando por las clases, librerías usadas o las vistas. Por otro lado, se explicará el desarrollo de la red de sensores, la configuración del servidor y la puesta en marcha de la red y la utilización de los datos en la aplicación.

4.1 Aplicación Android desarrollada en Kotlin

La aplicación se ha desarrollado siguiendo el patrón Modelo-Vista-Controlador (MVC). Este patrón distingue tres componentes que se encargan de la representación de la información y la interacción con el usuario. El modelo se encarga del manejo de los datos, la vista se encarga de la representación y el controlador se encarga de la lógica de la aplicación y de gestionar los eventos y comunicaciones con el usuario. En una aplicación Android, al referirse a la vista se habla de los ficheros XML que definen cómo se ve la vista con la que el usuario interactuará. En cuanto al modelo, podemos decir que son las clases que se encargan de obtener los datos que se mostrarán en la vista. Los datos podrán ser obtenidos desde una base de datos de la aplicación o mediante peticiones a servicios web. El modelo tendrá que gestionar el acceso a esos datos y ofrecérselos al controlador para que los muestre en la vista. Por último, los controladores serán las clases que harán de intermediario entre las vistas y el modelo. Son las clases que manejan toda la lógica de la aplicación. Normalmente son clases que extienden de “Activity” o de “Fragment”. Una “Activity” o actividad es un componente de la aplicación que contiene una pantalla. Para ser más concretos son las clases que se encargan de gestionar una pantalla o una parte de la aplicación, ya que puede que una actividad conste de más de una pantalla. Una aplicación en resumidas cuentas no es más que múltiples actividades que se vinculan entre sí. Un “Fragment”, fragmento en castellano, es aquel que representa y gestiona una parte del interfaz de usuario de una actividad. Hay actividades que constan de varias pantallas y

por ello se dividen en fragmentos para gestionar cada comportamiento o parte de la interfaz individualmente. Usar fragmentos en interfaces multipanel es la mejor manera de gestionar una actividad, ya que favorece la reutilización de código gracias a la modularización y ayuda a tener un código más limpio. Un fragmento es en esencia una subactividad que se puede usar en varias actividades.

Estructura del proyecto

Una vez explicado el patrón utilizado en la estructuración del proyecto vamos a ver la estructura de éste. En la Ilustración 34 se puede observar la estructura del proyecto de Android Studio. Vamos a explicar cómo se ha aplicado el patrón y qué corresponde a cada componente del patrón. La vista está recogida en la carpeta "res" que contiene los XML de las pantallas que forman la interfaz. El modelo, es decir los datos, se manejan en la carpeta "data", que contiene la lógica del acceso a los datos, y la carpeta "domain", que se encarga de solicitar los datos al recurso adecuado (servidor, base de datos o datos mockeados) y ofrecer comandos al controlador para realizar peticiones de unos datos específicos. Podríamos decir que las clases de la carpeta "domain" son las que ofrecen los datos al controlador y las de "data" las que acceden a los datos. Por último, los controladores son las clases de la carpeta "ui". Estas clases se encargan de gestionar la interfaz de usuario y son básicamente actividades, fragmentos y adaptadores.

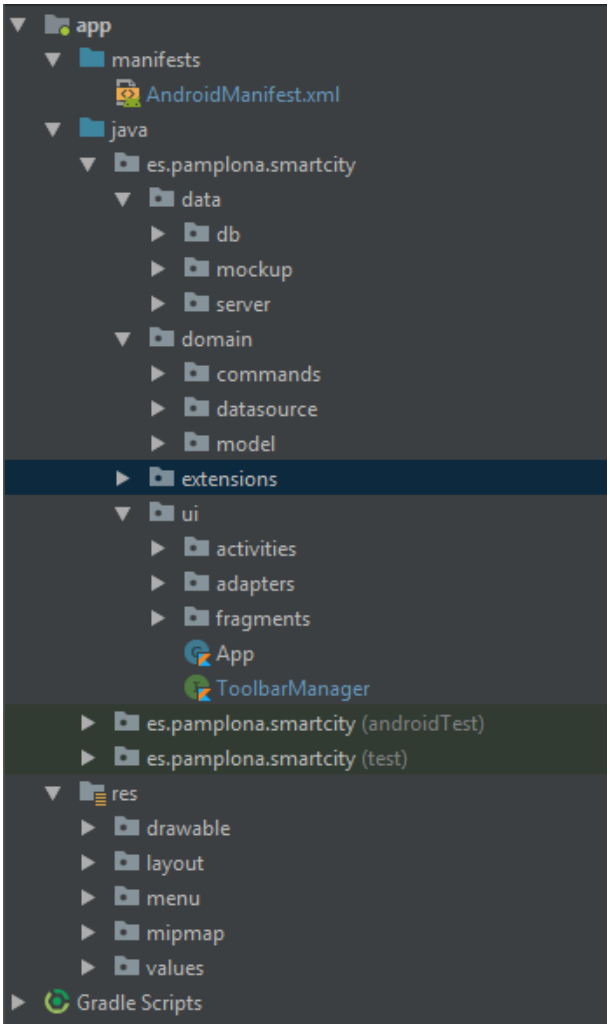

Ilustración 34. Estructura del proyecto de Android Studio

Ahora que ya se ha explicado los componentes que forman la estructura de la aplicación vamos a entrar más en detalle en cada una de ellas y analizar las clases que las forman. Para ello se seguirá la jerarquía de carpetas de la Ilustración 34. Para empezar, podemos ver el manifiesto que recoge los permisos que contendrá la aplicación y las actividades que forman la aplicación. El desarrollo en Kotlin se ha dividido en cuatro carpetas o paquetes: “data”, “domain”, “extensions” y “ui”.

Domain

Este paquete contiene la lógica del acceso a los datos, es decir, ofrece a los controladores los comandos para pedir unos datos en concreto y gestiona a que fuente debe pedírselos.

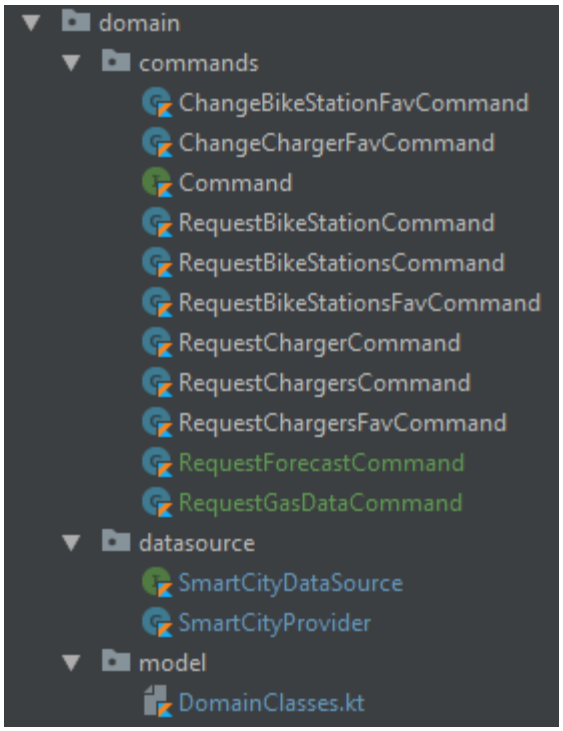

Ilustración 35. Clases del paquete Domain

En el paquete “commands” se presentan todos los comandos desarrollados. Un comando es la ejecución de una acción relacionada con datos. Un comando puede devolver algo o no, hay por ejemplo comandos que se usan para modificar datos en la base de datos (*ChangeCommand*) que no devuelven nada al ser usados y otros que son una petición de datos (*RequestCommand*) y que devuelven los datos solicitados. Los comandos definidos para esta aplicación son los siguientes:

- *ChangeBikeStationFavCommand*: Añade una estación de alquiler de bicicletas a favoritos si no está y la borra si está.
- *ChangeChargerFavCommand*: Añade un punto de recarga de vehículos a favoritos si no está y lo borra si está.
- *RequestBikeStationCommand*: Devuelve la información de la estación de alquiler de bicicletas cuyo identificador corresponda con el recibido.

- *RequestBikeStationsCommand*: Devuelve la información de todas las estaciones de alquiler de bicicletas.
- *RequestBikeStationsFavCommand*: Devuelve la información de las estaciones de alquiler de bicicletas favoritas del usuario.
- *RequestChargerCommand*: Devuelve la información del punto de recarga de vehículos cuyo identificador corresponda con el recibido.
- *RequestChargersCommand*: Devuelve la información de todos los puntos de recarga de vehículos.
- *RequestChargersFavCommand*: Devuelve la información de los puntos de recargar de vehículos favoritos del usuario.
- *RequestForecastCommand*: Devuelve la información meteorológica en la ciudad de Pamplona.
- *RequestGasDataCommand*: Devuelve la información sobre contaminación obtenida por la red de sensores.

Todas las clases del tipo *Command* solo tienen una función que se llama "execute" y que, como su propio nombre indica, ejecutan la acción. Los comandos no saben a qué fuente de datos deben solicitar los datos por lo que esta responsabilidad recae sobre las clases del paquete "datasource" o fuente de datos. Por lo tanto, lo que se ejecutará en la función "execute" del comando será la función correspondiente del *SmartCityProvider* que se explicará a continuación.

El paquete "datasource" contiene las clases que gestionan a que fuente se deben pedir los datos. En el interfaz *SmarCityDataSource* se definen todas las funciones, junto con sus parámetros de entrada y salida, que las clases que gestionen una fuente (*SmartCityDb*, *SmartCityMockup* y *SmartCityServer*) deberán implementar. Estas clases pertenecen al paquete "data", ya que son las que acceden directamente a los datos, y se explicarán más adelante. El número de funciones definidas en este interfaz coincide con el número de comandos definidos como cabe esperar. La clase *SmartCityProvider* gestiona las fuentes disponibles y realiza la petición a la fuente adecuada dependiendo de la función que sea. Una función puede tener más de una fuente por lo que esta clase también gestiona la prioridad de las fuentes. Por poner un ejemplo, para acceder a los datos sobre una estación de alquiler de bicicletas primero se intentaría obtener los datos de

la plataforma (que contiene los datos en tiempo real) y si no se puede se obtendrían de la base de datos (suponiendo que antes se hayan obtenido). Al menos así es como debería funcionar y se ha diseñado así para ello, pero en realidad, como esa plataforma todavía no se ha desarrollado, el orden que seguirá la aplicación se ha cambiado y primero intentará conseguirlos de la base de datos y si no es posible los obtendrá de los datos simulados.

Por último, en el paquete "model" se definen las clases de datos, "data class" en Kotlin. Estas clases definidas en *DomainClasses* son las que guardarán la información que usarán los controladores y que no tienen por qué ser iguales que las clases que modelen la información de la base de datos (*DbClasses*) o del servidor (*ServerClasses*) definidas en el paquete "data" que más adelante se comentarán.

Para entender esto se va a poner un ejemplo práctico. El orden para obtener los datos de las estaciones de alquiler de bicicletas es: primero pedir los datos al servidor y si no a la base de datos del teléfono. En la plataforma que se desarrollará se tendrá la información de dónde están situadas las estaciones de alquiler de bicicletas y su estado en tiempo real (bicicletas y huecos libres). Sin embargo, en la base de datos no nos interesa guardar la información del estado actual porque son unos datos en tiempo real y la base de datos no es la primera fuente. Un usuario que se conectó hace una semana a la aplicación cuando tenía internet no tiene por qué ver los datos de cómo se encontraba la estación en aquel momento si ahora no dispone de internet o no recibe respuesta de la plataforma. Se le estarían mostrando datos falsos, así que solo se le mostrarán los datos de donde está situada la estación. Por eso, la primera vez que se carguen los datos desde una petición a la plataforma hay datos sobre las estaciones que se almacenarán en la base de datos interna.

Una vez explicado esto, se entiende que las clases del dominio (*DomainClasses*) que usarán los controladores tendrán diferentes atributos que las clases de la base de datos (*DbClasses*) o del servidor (*ServerClasses*), por lo que será necesario mapearlas, como se explicará en el apartado del paquete "data". A continuación, se exponen las clases del dominio para poder compararlas luego con las de la base de datos y el servidor.

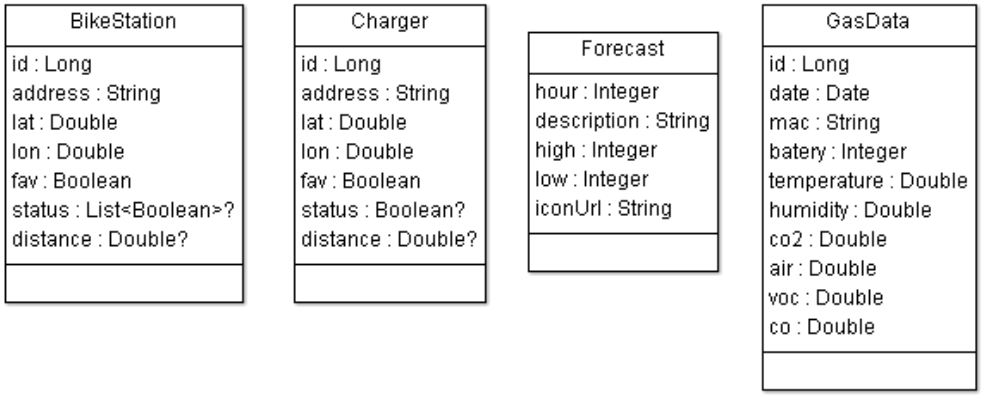

Ilustración 36. Diagrama de clases de DomainClasses

En este diagrama de clases se puede observar que algunas clases tienen atributos con tipos nullables (indicados con el signo "?"). Esto se debe al problema anteriormente citado y es que son datos que se deben obtener en tiempo real, por lo que si se obtienen desde la base de datos serán nulos.

Data

El paquete "data" contiene las clases de acceso a los datos. Se distinguen tres fuentes de datos diferentes: la base de datos SQLite, servidores y datos simulados o de demostración. Cada fuente tiene una clase principal (*SmartCityDb*, *SmartCityMockup* y *SmartCityServer*) que será llamada por la clase proveedora de datos (*SmartCityProvider*) comentada en el apartado anterior.

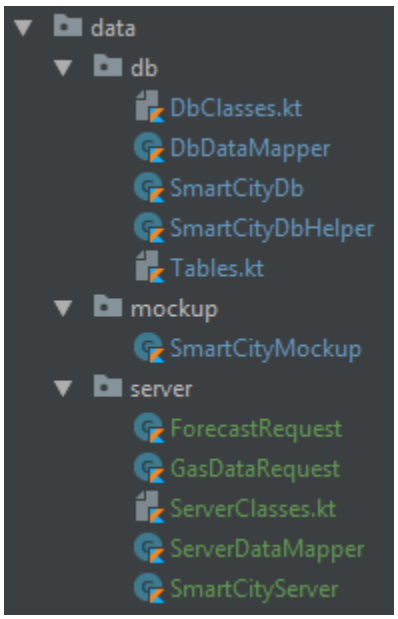

Ilustración 37. Clases del paquete Data

En el paquete "db" se gestiona la comunicación con la base de datos y la obtención, modificación e inserción de datos en esta. En *DbClasses* se definen las clases que permitirán mapear los datos de las tablas de la base de datos. Estas clases deberán ser mapeadas antes de poder ser usadas por los controladores. En la Ilustración 38 se muestra el diagrama de clases de las clases de *DbClasses.*

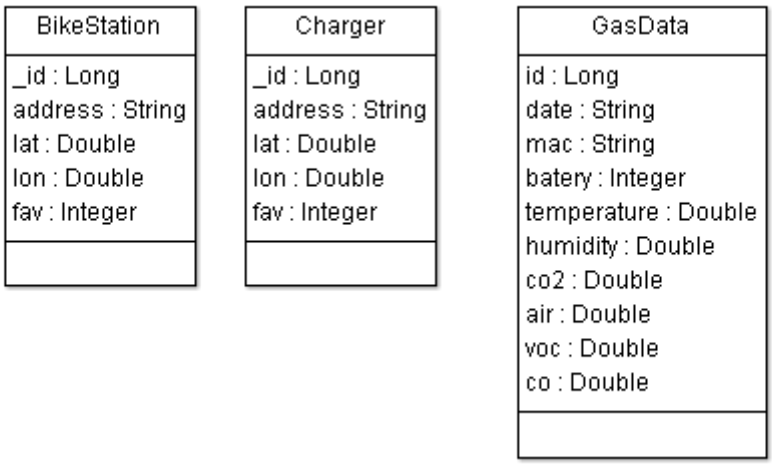

Ilustración 38. Diagrama de clases de DbClasses

La clase *DbDataMapper* es un "mapper" o mapeador, es decir, se encarga de recibir una clase y convertirla en otra. Esta clase contiene todas las conversiones de las clases definidas en *DbClasses* (las clases de la base de datos) a las ya explicadas *DomainClasses* (clases del dominio) y viceversa. Si lo que se desea es realizar una inserción en la base de datos, primero se mapeará una clase del dominio a una clase de la base de datos y después se realizará la inserción. Si por el contrario se trata de una extracción de datos, la consulta devolverá una clase de la base de datos y tendrá que ser mapeada a su equivalente del dominio para que el controlador haga uso de ella. En las Ilustraciones 39 y 40 se muestran los diagramas de secuencia genéricos para todas las inserciones y extracciones de datos y las clases que toman parte en estos procesos.

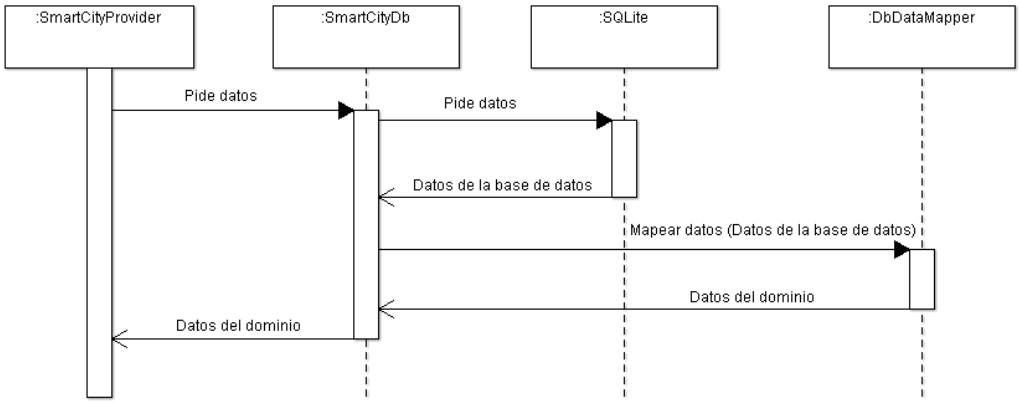

Ilustración 39. Diagrama de secuencia de la extracción de datos

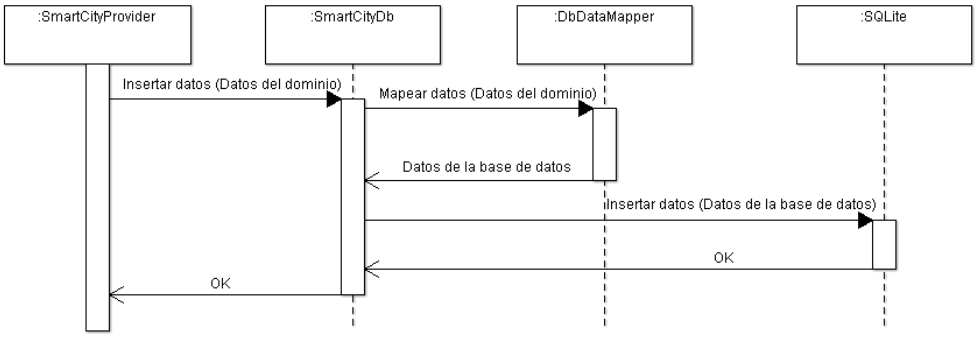

Ilustración 40. Diagrama de secuencia de la inserción de datos

Como aclaración, cuando en los diagramas se nombra "datos de la base de datos" se refiere a datos mapeados como clases de la base de datos, y cuando se dice "datos del dominio" a datos mapeados como clases del dominio. La clase *SmartCityDb* es la encargada de la comunicación con la base de datos SQLite. Esta clase implementa el interfaz *SmartCityDataSource* del paquete "domain" ya descrito. Es una de las fuentes usada por la clase *SmartCityProvider* para obtener datos. Permite extraer de la base de datos los datos almacenados en ella, actualizar las estaciones de alquiler de bicicletas y puntos de recarga favoritos e insertar los datos que se deben almacenar. Esta clase hace uso de la librería Anko [19], en concreto Anko SQLite, que simplifica el trabajo con esta base de datos. Ofrece métodos para la inserción, actualización y eliminación de datos de una formo sencilla y sin tener que usar los cursores de Android que hacen tan pesado el parseo de los resultados de las consultas. Esta librería ha sido desarrollada específicamente para Kotlin por sus mismos desarrolladores (Jetbrains). En la clase *Tables* se definen los nombres de las columnas de las tablas y, por último, la clase *SmartCityDbHelper* permite crear la base de datos y las tablas haciendo uso de las tablas definidas en *Tables*.

El segundo paquete de fuentes de datos es "mockup". Solo cuenta con una clase, *SmartCityMockup*, que en esta aplicación se usará para simular algunos de los datos que en un futuro la plataforma ofrecerá. Genera datos que se deberían obtener en tiempo real como la disponibilidad de los puntos de recarga o número de bicicletas disponibles en una estación. Al ser una fuente de datos, implementa la interfaz *SmartCityDataSource*.

Para finalizar con las fuentes de datos, se hablará de los datos obtenidos de un servidor. El paquete “server”, o más concretamente las clases incluidas en éste, son las encargadas de esta funcionalidad. En *ServerClasses* se definen las clases para recibir los datos de servidores. Al igual que los datos de la base de datos, necesitarán ser mapeados para ser usados por los controladores.

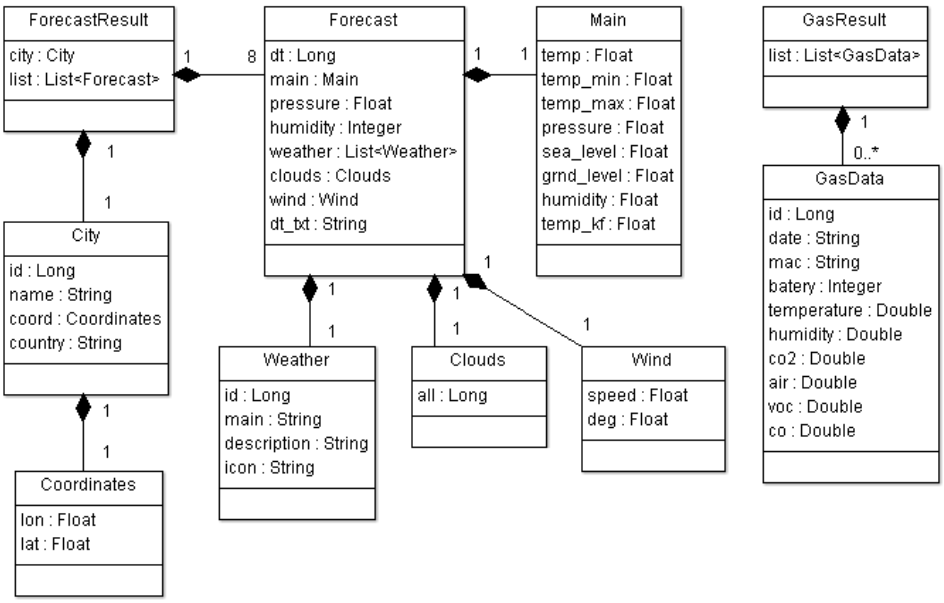

Ilustración 41. Diagrama de clases de ServerClasses

En la Ilustración 41 se pueden observar las clases que se usarán para guardar los datos de las peticiones a servidores recibidos en formato JSON. Se puede observar que estas clases sirven para datos de predicciones meteorológicas y para datos de la red de sensores. Si comparamos la clase Forecast con la clase Forecast del dominio (Ilustración 36) vemos que se usan muchos menos datos de los que se reciben, por lo que será necesario un mapeo. De esta tarea se encargará la clase *ServerDataMapper*. Esta clase realizará las conversiones de las clases del servidor (*ServerClasses*) a las clases del dominio (*DomainClasses*). Como por el momento no se envían datos a servidores, ya que la plataforma no está desarrollada, no es necesario la conversión a la inversa. En la Ilustración 42 se muestra el diagrama de secuencia genérico de una petición de datos a un servidor.

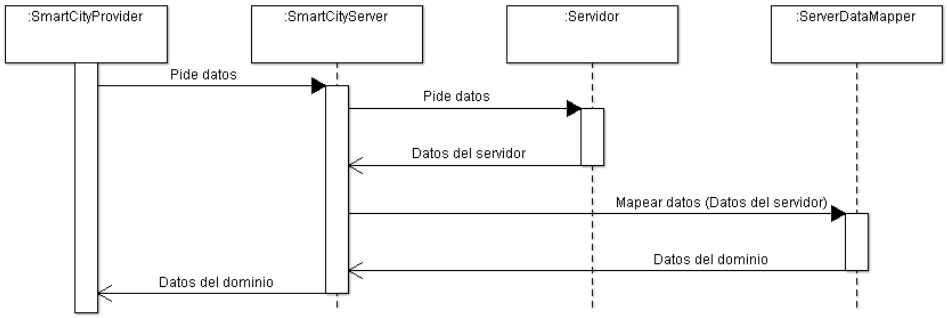

Ilustración 42. Diagrama de secuencia de la obtención de datos

SmartCityServer es la clase que actúa como fuente de datos de los servidores. Como las demás fuentes de datos implementa la interfaz *SmartCityDataSource* e interactúa con el proveedor de datos (*SmartCityProvider*). Esta clase se encargará de realizar la petición de datos al servidor adecuado, que en estos momentos son dos, el servicio web de la predicción de tiempo y el servicio web de los datos de los gases alojado en la Raspberry Pi. Las peticiones van encapsuladas en dos clases: *ForecastRequest* y *GasDataRequest*. *ForecastRequest* realizará una petición a la API de acceso libre OpenWeatherMap [20]. Este servicio web devolverá una respuesta con formato JSON con la predicción de tiempo cada 3 horas para las próximas 24 horas, es decir, 8 predicciones de las cuales al usuario solo se le mostrará la más próxima. La otra petición, *GasDataRequest,* realizará una petición al servicio web desarrollado en la Rapberry Pi, del que se hablará más adelante, y recibirá como respuesta los valores de gases recogidos por los sensores encapsulados en formato JSON.

UI

Este paquete está compuesto por los controladores de la aplicación. Estos controladores se encargan de la gestionar la interacción del usuario con la aplicación y de cargar los datos en las vistas. Los controladores se dividen en tres tipos de clases: actividades, fragmentos y adaptadores. Las actividades son clases que gestionan una funcionalidad de la aplicación y cada una de ellas puede estar formadas por una o más pantallas. Cuando una actividad requiere más de una pantalla se suele dividir en fragmentos. Los fragmentos son partes de una actividad que gestionan una pequeña funcionalidad o vista de una actividad. Para finalizar están los adaptadores. Los adaptadores son usados para vistas con forma de lista. Las listas están formadas por pequeñas partes de la interfaz que siguen una estructura,

mostrando la información de la misma manera para todos los elementos de lista. En la Ilustración 43 se puede ver la estructura del paquete "UI" dividido en tres paquetes que tienen los nombres de las clases de controladores recién comentadas. A diferencia de anteriores paquetes, en éste se comentará cada actividad y todos los fragmentos y adaptadores que lo componen, y no paquete por paquete.

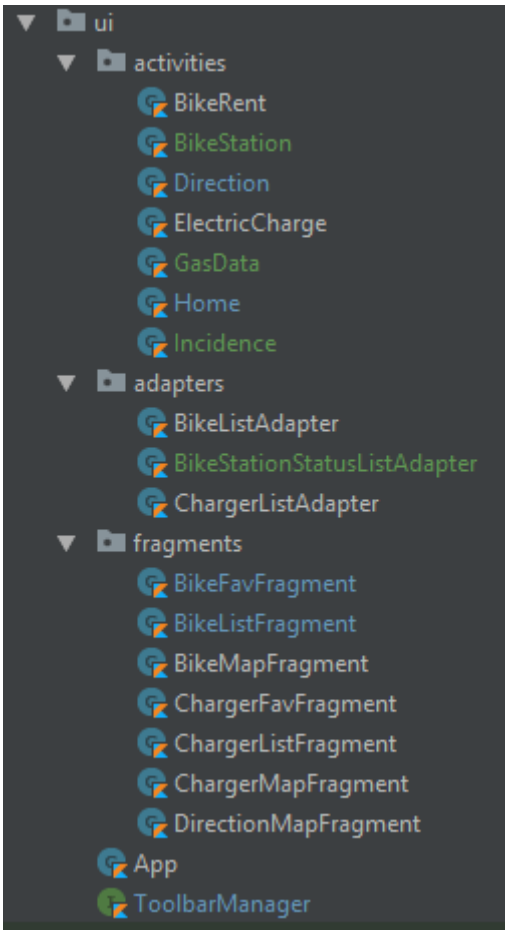

Ilustración 43. Clases del paquete UI

El paquete principal es denominado "activities". Es el paquete que contiene todas las actividades que forman la aplicación.

- *Home*

  La clase *Home* es la encargada de controlar la vista principal, es decir, la primera que el usuario ve cuando arranca la aplicación. Esta vista está formada por botones que llevan a cada una de las funcionalidades de la aplicación. Además de gestionar la interacción con estos botones, también se encarga de ejecutar el comando *RequestForecastCommand* que devuelve la predicción del tiempo y lo muestra en la parte superior de la vista. Hay que comentar que todos los comandos que ejecutarán las actividades o fragmentos se ejecutarán en segundo plano y que se usará la librería Anko [19] anteriormente citada para gestionarlos. De esta manera, en un primer momento se cargarán todas las partes de la vista que puedan ser cargadas y los datos se cargarán una vez se tenga el acceso a ellos. Esto agiliza la velocidad de la aplicación y da al usuario mejores sensaciones que si tuviera que esperar a los datos para ver la vista. Otro de los aspectos que se comentará aquí, y será igual para las siguientes vistas, es el uso de la interfaz *ToolbarManager*. Esta interfaz gestiona la barra superior ("Toolbar") y ofrece métodos para inicializarla con diferentes opciones. Las actividades la implementarán y cargarán la barra de herramientas que necesiten. Esta interfaz gestiona la interacción del

usuario con la barra, lo que exime a las actividades de su gestión. A continuación, se expone la vista que gestiona la clase *Home*.

Ilustración 44. Pantallazo de Home

- *BikeRent*

La clase *BikeRent* se encarga de la funcionalidad de alquiler de bicicletas. Esta actividad está formada por tres fragmentos: *BikeListFragment*, *BikeMapFragment* y *BikeFavFragment*. La clase *BikeRent* se encarga de gestionar los tres fragmentos y las transiciones entre ellos. Para ello cuenta con una barra de navegación inferior ("BottomNavigationView") con tres apartados: localizaciones, mapa y favoritos. Cada uno de estos apartados será respectivamente gestionado por los fragmentos recientemente nombrados. El controlador BikeRent por tanto lo tanto gestiona el contenedor de fragmentos que se sitúa entre la barra inferior y la superior, instanciando el fragmento adecuado según la opción seleccionada en la barra de navegación inferior. Este controlador implementa *ToolbarManager* para la gestión de la barra superior. Las vistas que gestionan los fragmentos se muestran en la Ilustración 45, Ilustración 46 e Ilustración 47.

- *BikeListFragment*

Esta clase se encarga de gestionar la vista en forma de lista de la información de las estaciones de alquiler de bicicletas. La vista está formada únicamente por una lista contenida en el elemento "RecyclerView", que es un widget más avanzado y flexible para

mostrar información en forma de lista que “ListView”. Para obtener la información de las estaciones ejecuta el comando *RequestBikeStationsCommand* en segundo plano y una vez reciba los datos se lo pasa al adaptador *BikeListAdapter* para que genere una entrada en la lista para cada estación de alquiler de bicicletas. Este adaptador cogerá la vista para los elementos de esta lista, la cargará con los datos de la estación y la inflará, es decir, la expondrá, al final de la lista (“RecyclerView”) del fragmento. El adaptador recibe como parámetros la lista de estaciones, la función que se debe realizar al hacer clic en un elemento de la lista y la función que debe realizar al hacer clic en el icono de favoritos. La función que se le pasa al hacer clic en el icono de favoritos es la ejecución del comando *ChangeBikeStationFav* que se ejecutará en segundo plano.

- *BikeMapFragment*

Esta clase se encarga de gestionar la vista en forma de lista. Para ello se hará uso de la API de GoogleMaps para Android [21]. Este fragmento realiza una petición a la API para cargar el mapa y una vez este cargado ejecuta el comando *RequestBikeStationsCommand* en segundo plano. Cuando recibe los datos, coloca un marcador en el mapa por cada estación de alquiler de bicicletas y ajusta el zoom del mapa al mayor nivel posible que encuadre todos los marcadores en él. Al hacer clic en un marcador, en la parte inferior derecha aparecen las opciones de que te lleve hasta allí y de abrir la ubicación en la aplicación de Google Maps.

- *BikeFavFragment*

Este fragmento muestra las estaciones favoritas del usuario en forma de lista. Ejecuta en segundo plano el comando *RequestBikeStationsFavCommand* para obtener la información y pasársela al adaptador de la lista, que en este caso en el mismo que el del fragmento *BikeListFragment*, es decir, el adaptador

BikeListAdapter. El adaptador funciona como se ha explicado en el apartado anterior

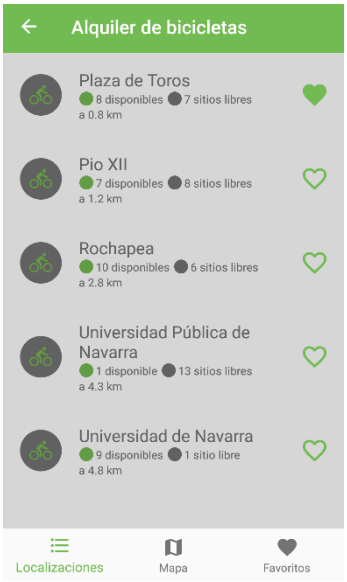

Ilustración 45. Pantallazo localizaciones BikeRent

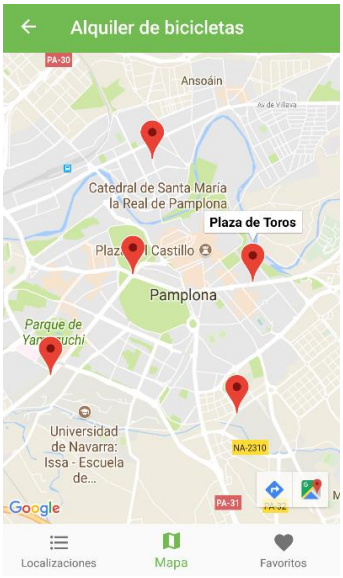

Ilustración 46. Pantallazo mapa BikeRent

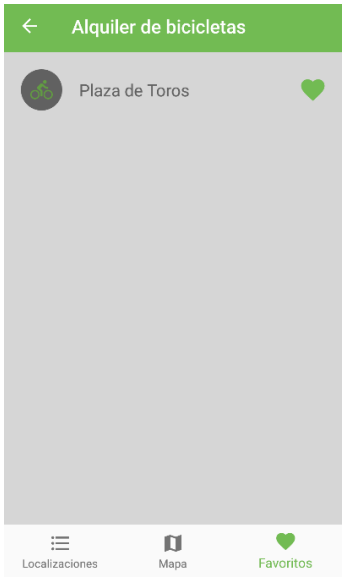

Ilustración 47. Pantallazo favoritos BikeRent

- *ElectricCharge*

Esta clase se encarga del control de la funcionalidad de los puntos de recarga de coches eléctricos. Al igual que la que gestiona el alquiler de bicicletas, está estructurada en una barra de navegación inferior, que gestiona el fragmento que se cargará en el contenedor, la barra superior y el contenedor. Cuenta con 3 fragmentos: *ChargerListFragment*, *ChargerMapFragment* y *ChargerFavFragment*. Este controlador también implementa la interfaz *ToolbarManager*.

- *ChargerListFragment*

Fragmento que se encarga de gestionar la vista de lista de la información relativa a puntos de recarga de coches eléctricos. Realiza la ejecución del comando *RequestChargersCommand* en segundo plano y le pasa los resultados al adaptador *ChargerListAdapter*. El funcionamiento de este adaptador será similar al de las bicicletas, lo que cambia es la vista de cada elemento de la lista. Cargará la información de cada punto de recarga en un elemento de la lista y lo "inflará" o expondrá al final de ella. Gestionará los clics en el elemento y en el icono,

ejecutando el comando *ChangeChargerFavCommand* en el caso de este último.

- *ChargerMapFragment*

Este fragmento gestiona la vista en forma de mapa de la información. Para ello usará la API de Google [21] como se ha comentado en la funcionalidad del alquiler de bicicletas. Realizará el mismo procedimiento para cargar el mapa, colocar los marcadores una vez se ejecute el comando *RequestChargersCommand* y ajustar el nivel de zoom.

- *ChargerFavFragment*

El último fragmento gestiona los puntos de recarga favoritos. Para ello ejecuta el comando *RequestChargersFavCommand* y utiliza el adaptador *ChargerListAdapter* de la misma manera que en el fragmento de lista para mostrar la información y gestionar la interacción con los elementos de la lista. Como se puede observar, el comportamiento de los tres fragmentos de la actividad *BikeRent* y los tres de *ElectricCharge* son prácticamente idénticos, a excepción de la carga de datos.

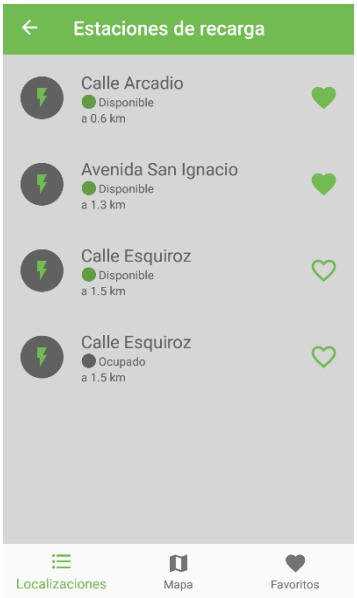

Ilustración 48. Pantallazo localizaciones ElectricCharge

Ilustración 49. Pantallazo mapa ElectricCharge

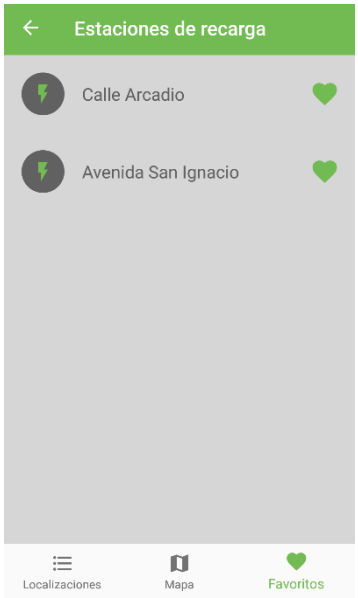

Ilustración 50. Pantallazo favoritos ElectricCharge

- *Direction*

Esta actividad gestiona una funcionalidad que de momento no se ofrece debido a que la plataforma que calculará las posibles rutas no está desarrollada. Aun así, se ha incluido en la aplicación para mostrar cómo debería ser. La aplicación consta de un contender con un único fragmento, *DirectionMapFragment*, que gestiona un mapa de Google. En el futuro deberá ejecutar un comando para obtener las mejores rutas valorando todas las opciones de desplazamiento posibles.

Ilustración 51. Pantallazo de Direction

- *Incidence*

Esta actividad ofrece la funcionalidad de crear un tique para el sistema de incidencias. Aunque de momento no se envíe el tique a la plataforma, se ha creado la vista que permitirá al usuario realizar esta acción. Para ello se cuenta con unas entradas para datos que permitan introducir un asunto y una descripción. En la Ilustración 52 se puede observar la vista que gestiona la clase *Incidence*.

Ilustración 52. Pantallazo de Incidence

- *GasData*

Para finalizar, se comentará la actividad que muestra la información de gases recogida por la red de sensores de la que se hablará más adelante. Esta clase gestiona una vista con tres partes: la barra de herramientas superior, que cuenta con “tabs” o botones, el “switch” o interruptor que permite cambiar el origen de los datos que se muestran y el gráfico que muestra los datos. Lo primero que realiza el controlador es la ejecución del comando *RequestGasDataCommand*. Una vez recibe los datos de los gases carga los datos por defecto, en este caso los de temperatura, en el gráfico. Mediante los “tabs” se puede escoger los valores de qué gas se quieren visualizar y mediante el “switch” situado debajo el origen de los datos. Esta vista está pensada para ser vista en horizontal para que los gráficos se vean mejor, aunque es posible verlos de manera horizontal. En las siguientes ilustraciones se muestran algunos gráficos de diferentes gases y origen de datos, aunque no se pretende analizar los datos ya que eso se realizará al hablar de la red de sensores.

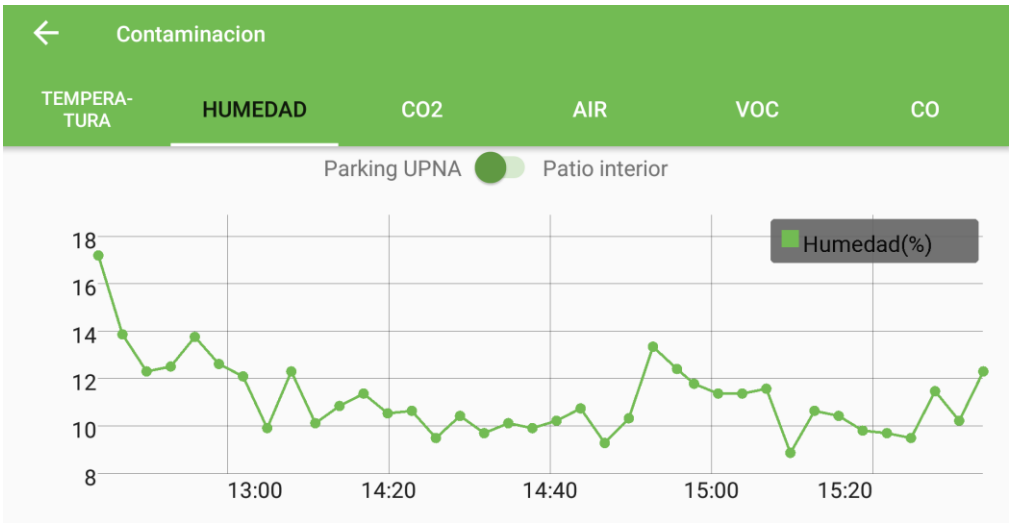

Ilustración 53. Pantallazo Humedad en parking UPNA

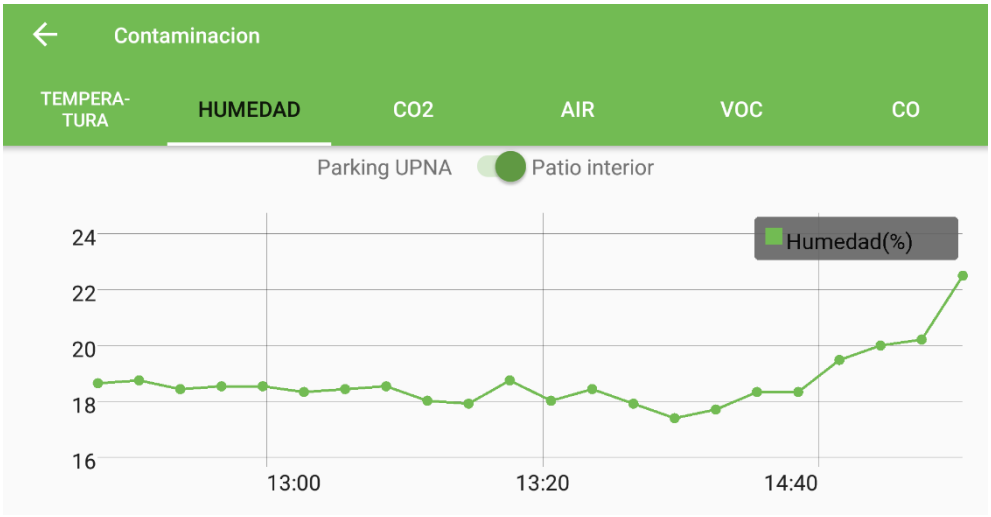

Ilustración 54. Pantallazo Humedad en patio interior

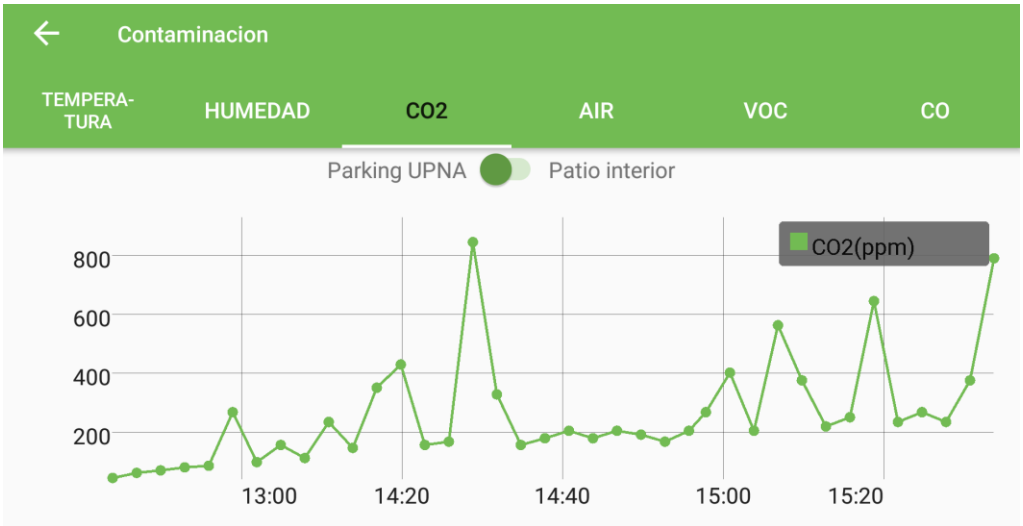

Ilustración 55. Pantallazo CO2 en parking UPNA

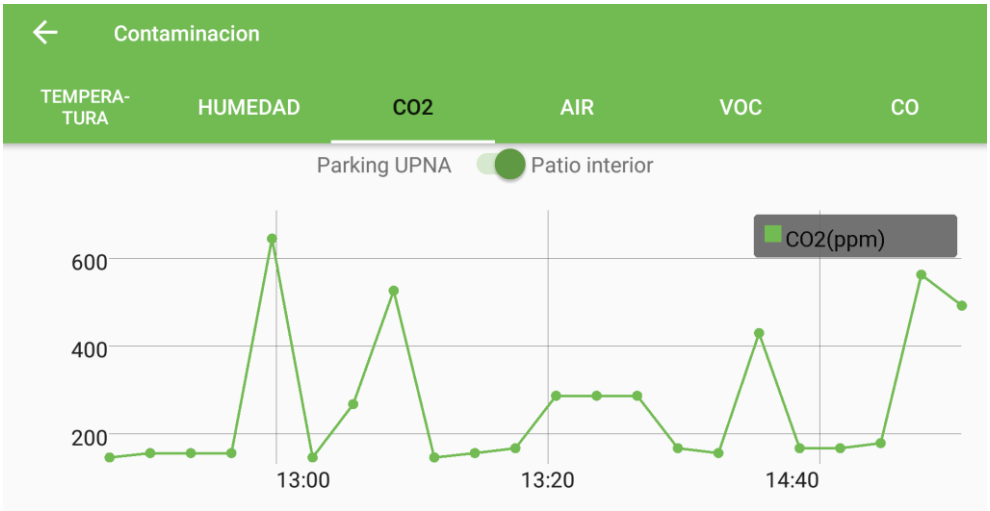

Ilustración 56. Pantallazo CO2 en patio interior

4.2 Red de sensores y servidor

El desarrollo de la red de sensores para la recolección de datos sobre la contaminación aérea tiene dos componentes muy diferentes y que conllevan desarrollos dispares. Por un lado, están los nodos de la red, que son placas Waspmote [15] que tienen acoplada una placa con sensores, tal y como se ha comentado en el capítulo 3 de diseño. Por el otro está el servidor, que es una Raspberry Pi con una arquitectura LAMP para poder acometer todas las funcionalidades que tiene que ofrecer. Para conectar el servidor a la red se usará una placa Waspmote programada solo para recoger los paquetes de la red, por lo que no contará con sensores de gases, pero sí con un módulo de comunicación ZigBee.

El programa que las placas Waspmote tienen instalado está estructurado en tres partes: la declaración de variables, la función “setup” y la función “loop”. La placa ejecuta el programa de la siguiente manera: cuando se enciende la placa o se reinicia se ejecuta la función “setup”, en la que se fijarán parámetros necesarios para la fase de “loop”, y una vez terminada esta fase se ejecuta la función “loop” en bucle como su propio nombre indica.

En la declaración de variables se definen parámetros como los tiempos de “loop” y de “setup”, las resistencias que se les deberán poner a los sensores, el ID de la red o la dirección MAC de la placa receptora conectada al servidor, es decir, la dirección de destino de los paquetes. El tiempo de “loop” indica el tiempo que pasará entre las ejecuciones de la función

“loop”. En este caso significa el tiempo que pasa desde que los sensores recogen los datos y se los envían al servidor hasta que se vuelvan a recoger los próximos datos. El tiempo de “setup” es el tiempo que pasa desde que se finaliza el “setup” hasta que se ejecuta la función en bucle por primera vez. Este tiempo es necesario para que los sensores cojan las temperaturas óptimas para realizar mediciones. Los tiempos de “setup” y “loop” se han fijado en 2 y 5 minutos respectivamente. Los valores de las resistencias se han fijado basándose en las fichas técnicas de los sensores. Los sensores de temperatura, humedad y dióxido de carbono (CO_2) no necesitan que se les fije ningún valor de resistencia. Sin embargo, los sensores de contaminantes de aire, de compuestos orgánicos volátiles y el de monóxido de carbono (CO) sí que requieren fijar este parámetro y se han fijado en 50kΩ, 50kΩ y 15kΩ respectivamente. Además, la ganancia de todos los sensores se ha establecido en 1, el valor por defecto. Tanto las resistencias cargadas en los sensores como la ganancia sirven para calibrar los sensores. En este caso, al no disponer de equipo para calibrar los sensores, se ha optado por utilizar los valores recomendados o por defecto a pesar de que los datos obtenidos puede que no sean exactos.

En la función “setup” se inicializan todos los sensores y antenas. Para ello hay que indicarle a la placa Waspmote qué placa de sensores tiene conectada, en este caso la de gases. Una vez hecho esto se deben configurar los sockets en los que se conectará cada sensor cargando las resistencias y ganancias definidas en la declaración de variables. Una vez configurados todos los parámetros que conciernen a los sensores, es necesario configurar los parámetros de la red. Se debe activar la antena de ZigBee y configurar el ID de la red y el canal usado, ya que cuenta con 16 canales. Lo último que hace es guardar la MAC de la antena en una variable global, ya que es uno de los parámetros que llevará el paquete que se mandará y así no se tiene que calcular en cada bucle. Una vez finalizada la ejecución de la función de “setup” esperará el tiempo de “setup” antes de empezar con el bucle.

En la función “loop” se define lo que realizará la placa cada x tiempo. En este caso se quiere que recoja los datos de los sensores, cree una trama de red y la envíe al servidor. Primero recoge los valores de todos los sensores

y los guarda en variables. Después crea un paquete ZigBee denominado “packetXBee” y se configura el modo de envío, en este caso “unicast”, es decir, el paquete va dirigido a un único receptor, el ID del paquete, la MAC del emisor, la MAC del receptor y los datos que contiene el paquete. Los datos que contendrá el paquete es una frase o “string” estructurada para que el servidor la pueda tratar. Una vez tiene el paquete listo lo envía y espera el tiempo de “loop” para volver a empezar el proceso. La placa ejecutará esta función hasta que se apague o se quede sin batería.

Una vez analizado el comportamiento de los nodos de la red se procede a comentar el funcionamiento del servidor. El servidor tiene conectada una Waspmote a través de una conexión de USB a la entrada microUSB de la placa como se muestra en la Ilustración 57.

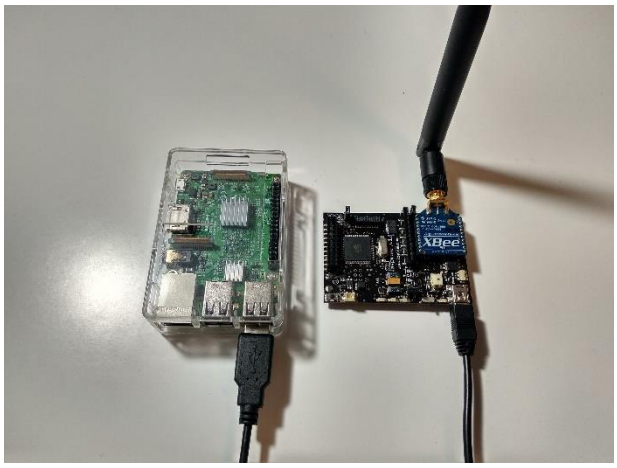

Ilustración 57. Conexión entre servidor y red de sensores

El programa con el que cuenta este nodo es diferente al que tienen los demás nodos de la red. Este programa en la función “setup” configura los parámetros de la red y activa la antena ZigBee que será la encargada de recibir los paquetes. En este caso el tiempo de “setup” es de 5 segundos, tiempo muy inferior al que necesitaban las placas con sensores y que sirve para que el módulo de comunicación se inicialice. La función de “loop” se ejecutará constantemente, es decir, con un tiempo de “loop” de 0 segundos. Estará constantemente esperando paquetes y una vez reciba uno cogerá los datos del paquete que contienen los valores de los sensores y los imprimirá por la salida USB. De esta manera, los datos podrán ser recogidos por el servidor a través del puerto de serie. Cuando termine de tratar un paquete esperará al siguiente sin demora como ya se ha comentado.

La placa Raspberry ha tenido que ser configurada previamente para poder recoger estos datos a través del puerto de serie o USB. Lo primero que se hizo fue instalar el sistema operativo Raspbian Pixel en la placa. Para ello se le acoplo una tarjeta microSD con el almacenamiento suficiente para el sistema operativo y los programas y funcionalidad que se debían instalar. Una vez listo el sistema operativo se procedió a la instalación de los demás componentes de la arquitectura LAMP, es decir, un servidor Apache, un gestor de base de datos MySQL y un entorno de desarrollo PHP, basándose en algunos tutoriales de cómo montar una arquitectura de este tipo en una placa Raspberry Pi [22] [23]. El orden de instalación fue primero instalar el servidor Apache, luego las librerías de PHP (en su versión 7.0) y por último el gestor de base de datos MySQL y la herramienta phpMyAdmin que permite gestionar MySQL gráficamente a través de internet. Esta última herramienta hace más amigable la creación de bases de datos y tablas, así como las modificaciones en ellas o la visualización de datos, que si no se debería realizar a través de una terminal.

Una vez llegado a este punto ya se tiene todo el entorno de desarrollo preparado para que el servidor realizara las tareas que debía acometer. Se tiene por lo tanto un servidor Apache que permitía comunicarse con el servidor desde el exterior y que se levantaba, es decir, se iniciaba, cuando se arrancaba la Raspberry. Sabiendo cómo eran las tramas que enviaban los nodos y que información contenían, se debía crear una base de datos con una tabla que almacenara la información. El diseño de la tabla es simple: un id autoincremental como clave primaria, la MAC del nodo para poder saber de dónde procede la información, la fecha y hora de la recogida de los datos, la batería restante del nodo y un campo por cada valor de sensor. En la Ilustración 58 se muestra la estructura de la tabla obtenida de la herramienta PHPMyAdmin.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra
1	**id**	int(11)			No	*Ninguna*		AUTO_INCREMENT
2	**date**	timestamp			No	CURRENT_TIMESTAMP		
3	**mac**	varchar(25)	utf8_general_ci		No	*Ninguna*		
4	**batery**	int(11)			No	*Ninguna*		
5	**temperature**	float			No	*Ninguna*		
6	**humidity**	float			No	*Ninguna*		
7	**co2**	float			No	*Ninguna*		
8	**air**	float			No	*Ninguna*		
9	**voc**	float			No	*Ninguna*		
10	**co**	float			No	*Ninguna*		

Ilustración 58. Estructura de tabla Air

Ahora se debía poblar la base de datos, más concretamente la única tabla creada, con los datos que se recibían a través del USB. Para ello se realizó un programa en el lenguaje de programación PHP que se ejecutaría en el servidor Apache. Se utilizó la librería PHP Serial [24] para abrir una conexión con el puerto serie y configurar los parámetros:

- Puerto: /dev/ttyUSB0
- Tasa de baudios por segundo: 38400
- Bits de paridad: 0
- Número de bits de datos: 8
- Bits de stop: 1

El puerto indica en qué puerto está conectada la Waspmote, la tasa de baudios indica la velocidad a la que se transmite la información, el no tener bits de paridad indica que no se detectaran errores de paquetes, el número de bits de datos indica cuantos bits contienen datos como su propio nombre indica y por último el “stop bit” indica que se ha transmitido un byte. Una vez abierta la conexión con el puerto de serie entra en un bucle en el que se quedará bloqueado hasta que lea algo del puerto de serie. Si consigue leer algo entonces parseará los datos obtenidos para saber a qué gas o parámetro corresponde cada valor. Una vez lo haya descifrado, abrirá una conexión con la base de datos creada e insertará en la tabla “air” los datos en sus correspondientes campos. Para ello ejecutará la sentencia SQL de inserción que se expone a continuación:

INSERT INTO air (mac,batery,temperature,humidity,co2,air,voc,co) VALUES (?,?,?,?,?,?,?,?)

Los campos “id” y “date” se completan automáticamente por lo que no es necesario asignarles un valor. Los símbolos de interrogación serán sustituidos por los valores parseados recibidos a través del puerto de serie. Una vez ejecutada la sentencia volverá a esperar un nuevo mensaje del puerto USB. Este programa se empezaría a ejecutar sin pausa una vez levantado el servidor Apache.

Ahora ya se tenía una red de sensores que mandaba datos periódicamente a la placa conectada al servidor a través de ZigBee, la placa transmitía los datos al servidor a través del puerto USB y el servidor se encargaba de recogerlos y almacenarlos en la base de datos. Solo se debía desarrollar el servicio web que permitiera a la aplicación recoger los datos de la base de datos MySQL. Para ello se creó un programa también desarrollado en PHP que ofreciera este servicio. El programa debía devolver los datos de los gases en el formato JSON al ser invocado por una petición HTTP por parte de la aplicación. Primero abría una conexión con la base de datos y realizaba la siguiente consulta SQL:

```
SELECT * FROM air ORDER BY id ASC
```

Esta consulta devuelve todos los datos de la tabla “air” ordenados por el “id”. Para terminar parseaba el resultado al formato JSON y lo devolvía. De esta manera cuando la aplicación realiza la petición *GasDataRequest* obtiene los datos recibe los datos obtenidos y los puede mostrar gráficamente como se ha visto en el apartado anterior.

5. Análisis de resultados

Una vez visto cómo se recogen los datos en los nodos, se almacenan en el servidor y se muestran en la aplicación es necesario valorar los resultados obtenidos. Para empezar, es necesario decir que se reunieron datos en dos localizaciones muy dispares. Algunos datos fueron tomados en un parking de la Universidad Pública de Navarra y otros en el patio interior de una casa, por lo que no hay carreteras. Obviamente, uno de los grandes problemas del medio ambiente a día de hoy es la cantidad de vehículos que hay y los gases contaminantes que estos expulsan. Este factor será determinante en los resultados, ya que en uno de los sitios en los que se hizo la medición los coches no pasan y por el otro pasan constantemente.

Los datos, cómo se ha visto, se obtienen de los valores devueltos por los sensores. Los sensores devuelven un valor en voltaje siempre, por lo que es necesario hacer una conversión para obtener el valor en la medida que buscamos. Estos valores suelen venir dados por funciones lineales, exponenciales o logarítmicas. Además, algunos de los sensores requieren una calibración para su óptimo funcionamiento. Estas calibraciones requieren equipos muy sofisticados y generalmente caros que crean unas condiciones en el entorno muy concretas. Se suelen realizar en entornos cerrados para su correcta realización. Durante las pruebas realizadas no se ha llevado a cabo una calibración de los sensores por las razones ya comentadas, así que los datos tomados tendrán desviaciones de los valores reales. En su lugar se han usado datos de calibraciones obtenidos de la empresa que fabrica las placas (Libelium). A pesar de que son datos de la calibración de los mismos tipos de sensores, cada sensor es diferente y requiere su calibración. A continuación, se van a analizar y comparar los datos obtenidos en las dos ubicaciones.

- Temperatura

Los resultados de temperatura son cuanto menos curiosos. En el parking de la UPNA los valores obtenidos fueron erróneos y marcaba entre 10 y 20 grados más de los que realmente hacía. Además, se puede apreciar unos altibajos importantes que distan mucho de la realidad. Esto posiblemente se deba a que se situó la placa a pleno sol en las horas más calurosas del día y el sensor se sobrecalentó dando lecturas erróneas. Sin embargo, en los datos obtenidos en el patio interior se puede apreciar una temperatura en torno a los 23-24º C. Estos datos pareces más reales que los primeros y seguramente se deba a que el sensor estaba a la sombra hasta que más o menos a las 15:00 le empezó a dar un poco el sol. Aun así, se puede apreciar que los valores se asemejan más a la realidad y el hecho de obtener datos erróneos en el parking de la universidad se debe más a una mala colocación de la placa que a un fallo del sensor. Debido a que las baterías se habían prácticamente agotado en las 3 horas que se recogieron datos, y viendo que el error no venía del sensor si no de la colocación de éste, se decidió no repetir la prueba para poder sacar las conclusiones oportunas de los datos recogidos, entre las que se encuentra la importancia de la colocación de los nodos.

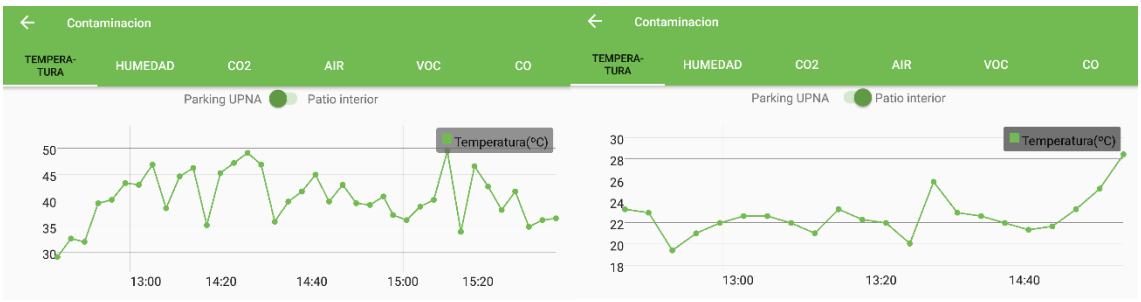

Ilustración 59. Comparación valores de temperatura

- Humedad

Los datos de humedad, a diferencia de los de temperatura, parecen más reales. En el caso del sensor situado en el parking se aprecia una humedad media entorno al 11%. Teniendo en cuenta que la placa estaba al sol bien podrían ser reales, aunque al igual que los de temperatura, tienen pequeños altibajos que varían un 2% por encima y debajo de la

media. Los datos obtenidos en el patio interior se mantienen bastante lineales y en torno al 19% de humedad. Al ser una zona en sombra y en un día caluroso a principios de mayo, tiene sentido que los datos sean los obtenidos.

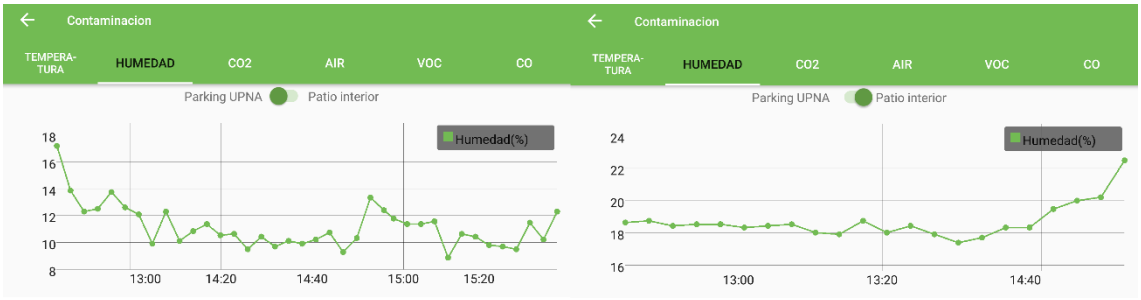

Ilustración 60. Comparación valores de humedad

- CO_2

Los valores de CO_2 se miden en ppm, es decir, partes por millón. Con esta unidad de medida se mide la concentración de una sustancia en un conjunto. En este caso representa la cantidad de unidades de CO_2 que hay por cada millón de unidades de aire, entre los que puede haber otros gases y sustancias. Según los datos de un estudio [25] se puede considerar aire limpio de CO_2 si no supera las 440 ppm y la contaminación de esta sustancia en las ciudades ronda las 700 ppm. Como se puede observar en la Ilustración 61, ambas gráficas contienen bastantes picos. En el caso del parking es más lógico ya que se puede deber al movimiento de vehículos en el parking. Los picos coinciden con las 14:30 y las 15:00-15:30 que son las horas en las que la gente sale a comer o del trabajo. Sin embargo, los picos del gráfico del patio dan la sensación de tener algún error, debido a que tiene picos bastante elevados para no estar muy cerca de carreteras.

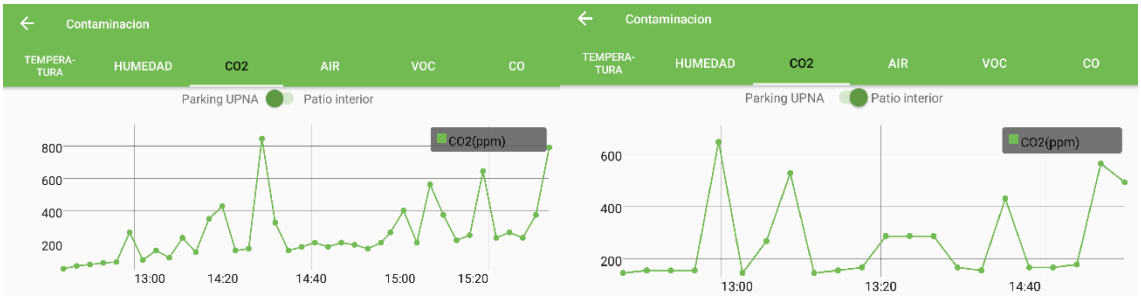

Ilustración 61. Comparación valores de CO2

- Contaminantes de aire

Este sensor capta información de varios gases que contaminan el aire, en concreto el metilpropano (C_4H_{10}), el etanol (CH_3CH_2OH), el hidrogeno (H_2), el monóxido de carbono (CO) y el metano (CH_4). Este dato, al igual que el anterior, se mide en partes por millón (ppm). Ambas gráficas tienen una tendencia bastante lineal y rondan los mismos valores en torno a las 83 ppm. En el caso del parking alcanza algún que otro pico superior a las 85 ppm.

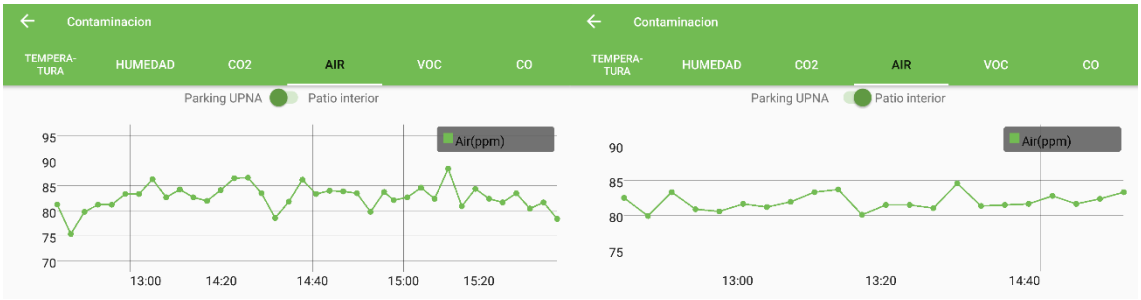

Ilustración 62. Comparación valores contaminantes de aire

- Compuestos orgánicos volátiles

Los valores de este sensor también se miden en partes por millón. En ambas ubicaciones los resultados ofrecen lecturas muy parecidas y lineales que rondan las 3000 ppm. Según un estudio de la Universidad del Norte de Arizona [26], un nivel admisible de VOC es de 0.75 ppm. En ambas gráficas los niveles están muy por debajo de estos niveles y las mediciones se mantienen bastante estables en torno a las 0.1 ppm.

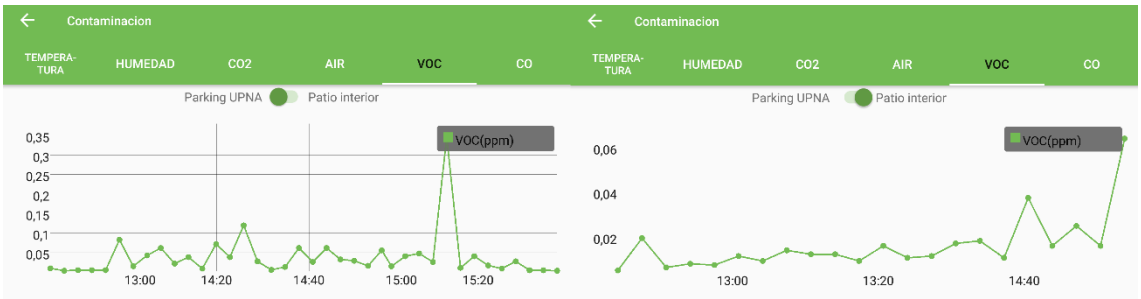

Ilustración 63. Comparación valores VOC

- CO

Para finalizar tenemos los datos del monóxido de carbono. También medido en ppm, se considera [25] valor límite o umbral las 30 ppm. En

ninguna de las mediciones se observan valores que se acerquen a esta cifra. En este caso las diferencias entre ambas ubicaciones son bastante notables y esto se debe a que el monóxido de carbono es uno de los gases principales que los vehículos con motores de combustión expulsan por los escapes. En el caso del parking las mediciones rondan las 2-3 partes por millón en una gráfica muy lineal. En las mediciones en el patio interior los valores rondan las 0,00004 ppm, y al igual que en el parking se muestra una gráfica muy lineal. En este apartado la diferencia es notable entre una ubicación y otra.

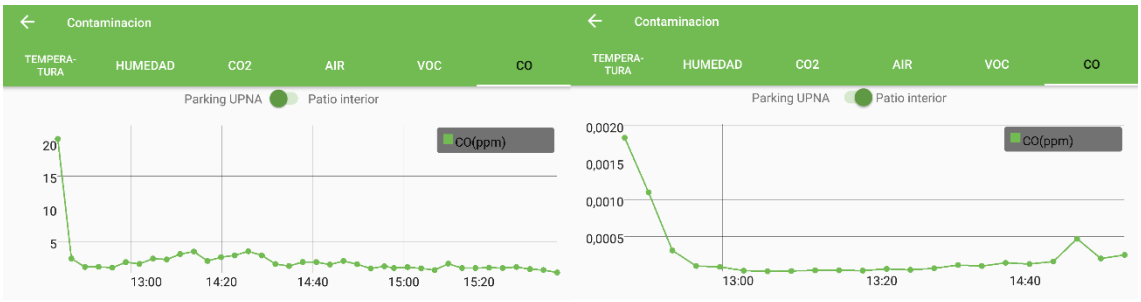

Ilustración 64. Comparación valores de CO

6. Conclusiones

Como se ha podido observar a lo largo del documento el trabajo ha contado con partes muy diferentes entre sí, que han requerido el uso de herramientas y lenguajes muy diferentes para su desarrollo.

La elección de Kotlin como lenguaje de programación para la aplicación Android fue todo un acierto debido a la comodidad y posibilidad que ofrece al desarrollar. Viniendo de experiencias anteriores en desarrollos para este sistema operativo usando Java como lenguaje de programación, he de decir que me ha sido bastante sencilla la adaptación al nuevo lenguaje y a medida que vas descubriendo las facilidades que ofrece respecto a Java te vas sintiendo más cómodo y reduciendo el tiempo de desarrollo necesario para finalizar funcionalidades de la aplicación. Por poner un ejemplo, el hecho de no tener que hacer uso de ningún bucle “for” o “while” ha sido uno de los cambios más notorios a la hora de desarrollar. Las posibilidades que ofrece la librería Anko [19], de las cuales solo he llegado a profundizar en unas pocas, han hecho que el acceso a los elementos de las vistas o el trabajo con la base de datos SQLite haya resultado mucho más fácil y comprensible. Otro de los factores importantes en el desarrollo de la aplicación ha sido el libro de Antonio Leiva sobre Kotlin [11]. En él he podido aprender, además de lo básico del lenguaje Kotlin, a estructurar correctamente una aplicación. La sencillez con la que explica las funcionalidades básicas de Kotlin y las comparaciones con Java han ayudado mucho a esa transición de la que se hablaba anteriormente de Java a Kotlin.

En cuanto al contenido de la aplicación me quedo con la sensación de haber cumplido a medias con lo que debería ser la aplicación en un futuro. Me hubiese gustado trabajar con los datos en tiempo real de las estaciones de bicicletas, puntos de recarga, rutas de como desplazarse, etc. Me quedo con que todo lo desarrollado ha quedado preparado para una conexión con la plataforma que en un futuro no muy lejano estará disponible para ofrecer

todos estos datos. La correcta estructuración de la aplicación hace que la lógica de los datos, los controladores y las vistas estén bien diferenciadas y sea intuitivo donde se deberían introducir las nuevas funcionalidades. Además, creo que hay más funcionalidades que se podrían incluir en esta aplicación relacionadas con una ciudad inteligente y que deberían ser reunidas en una única aplicación. En el apartado 1.4 del estado del arte se citan al menos seis aplicaciones con diferentes funcionalidades y todas ellas relacionadas con la ciudad, que bien se podrían reunir en una sola facilitando a los usuarios el acceso a toda la información sin tener que tener seis o más aplicaciones diferentes instaladas en su dispositivo.

En cuanto a la colaboración ciudadana, creo que se debería hacer un esfuerzo en plantear correctamente el sistema de tiques y ofrecer la manera más sencilla a los usuarios para comunicarse con el ayuntamiento. Algún tipo de incentivo, como ya se ha comentado anteriormente, para que los ciudadanos ayuden a mejorar la ciudad debería ser un factor a plantearse en mi opinión.

Por lo que a la red de sensores respecta, creo que se deberían cambiar algunas cosas antes de implantar un sistema como este en la ciudad. Para empezar los sensores usados deberían ser calibrados para que las lecturas obtenidas fueran lo más reales posibles. Otro de los problemas son las baterías de las placas, ya que en 3 horas de uso su carga se redujo en un 60%, por lo que habría que utilizar un sistema alternativo como la energía solar o baterías con mayor capacidad para que los nodos se mantengan activos la mayor cantidad de tiempo posible. La conectividad de la red y la comunicación usada también debería ser revisada en el caso de querer implantar una red más extensa y con obstáculos de por medio. Además, creo que sería necesario algún tipo de tratamiento en la plataforma que los almacenara para sacar unas conclusiones más extensas de cada valor recogido y que expertos en la materia pudieran realizar estudios de la contaminación en diversas zonas de la ciudad. Estos datos, a pesar de que en este proyecto se muestran en la aplicación móvil, me parecen mucho más útiles para el propio ayuntamiento que para los usuarios y sería más conveniente, como se ha comentado, que los datos fueran analizados y tras un estudio ser presentados a los ciudadanos como tal, es decir, como un

estudio, y no en tiempo real en la aplicación. El servidor, la Raspberry Pi, como es de imaginar, no sería capaz de manejar grandes cantidades de datos y procesarlos porque sus especificaciones técnicas no lo permiten, pero la plataforma desarrollada sí que tendrá la capacidad de acometer estas funciones y sería beneficioso hacer uso de ellas.

7. Bibliografía y referencias

[1] Proyecto STARDUST, http://stardustproject.eu/ [Accedido en mayo 2018]

[2] Estrategia Smart City Pamplona, http://www.pamplona.es/verDocumento/verdocumento.aspx?idDoc=264537 [Accedido en mayo 2018]

[3] Horizonte 2020, https://eshorizonte2020.es/ [Accedido en mayo 2018]

[4] Cuotas versiones de Android, https://developer.android.com/about/dashboards/ [Accedido en mayo 2018]

[5] Material Design, https://material.io/ [Accedido en mayo 2018]

[6] Aplicación de robo de bicicletas, http://www.diariodenavarra.es/noticias/navarra/pamplona-comarca/pamplona/2018/05/14/nueva-app-pamplona-con-fotos-bicis-robadas-que-policia-recuperado-591461-1702.html [Accedido en mayo 2018]

[7] Aplicación de accesibilidad, https://pamplonaactual.com/pamplona-conocera-a-traves-de-una-aplicacion-las-incidencias-en-el-entramado-urbano-que-impiden-la-accesibilidad/ [Accedido en mayo 2018]

[8] Puntos de recarga Pamplona, http://www.noticiasdenavarra.com/2018/01/18/vecinos/pamplona/pamplona-dispondra-de-40-puntos-de-recarga-para-coches-electricos [Accedido en mayo 2018]

[9] Android, http://androidos.readthedocs.io/en/latest/ [Accedido en mayo 2018]

[10] Cuotas de mercado Android, https://www.kantarworldpanel.com/global/smartphone-os-market-share/ [Accedido en mayo 2018]

[11] Antonio Leiva, “Kotlin for Android Developers”, 1st edition, 2016.

[12] Basecamp 3, https://m.signalvnoise.com/how-we-made-basecamp-3s-android-app-100-kotlin-35e4e1c0ef12 [Accedido en mayo 2018]

[13] Raspbian, https://www.programoergosum.com/cursos-online/raspberry-pi/232-curso-de-introduccion-a-raspberry-pi/instalar-raspbian [Accedido en mayo 2018]

[14] Moqups, https://moqups.com/ [Accedido en mayo 2018]

[15] Waspmote, http://www.libelium.com/v11-files/documentation/waspmote/waspmote-technical_guide_eng.pdf [Accedido en mayo 2018]

[16] Meshlium, http://www.libelium.com/v11-files/documentation/mesh_extreme/meshlium-technical_guide_eng.pdf [Accedido en mayo 2018]

[17] MySQL Ranking, https://db-engines.com/en/ranking [Accedido en mayo 2018]

[18] Raspberry Pi 3 Model B, https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ [Accedido en mayo 2018]

[19] Anko, https://github.com/Kotlin/anko [Accedido en mayo 2018]

[20] OpenWeatherMap, https://openweathermap.org/ [Accedido en mayo 2018]

[21] Google Maps Android API, https://developers.google.com/maps/documentation/android-api/?hl=es-419 [Accedido en mayo 2018]

[22] Instalación LAMP, https://www.atareao.es/tutorial/raspberry-pi-primeros-pasos/lamp-raspberry-pi/ [Accedido en abril 2018]

[23] Instalación LAMP, https://geekytheory.com/tutorial-raspberry-pi-15-instalacion-de-apache-mysql-php [Accedido en abril 2018]

[24] Librería PHP Serial, https://github.com/Xowap/PHP-Serial [Accedido en abril 2018]

[25] Estudio CO y CO_2, http://analizador-gases.es/test-1/ [Accedido en mayo 2018]

[26] Estudio NAU, https://www7.nau.edu/itep/main/eeop/docs/airqlty/AkIAQ_VolatileOrganicCompounds.pdf [Accedido en mayo 2018]